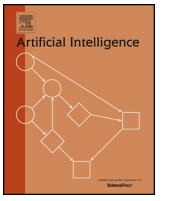
Artificial Intelligence 321 (2023) 103937

Contents lists available at ScienceDirect

Artificial Intelligence

journal homepage: www.elsevier.com/locate/artint



Spatial state-action features for general games

Dennis J.N.J. Soemers [a,*], Éric Piette [b,a], Matthew Stephenson [c], Cameron Browne [a]

[a] Maastricht University, Department of Advanced Computing Sciences, Paul-Henri Spaaklaan 1, 6229 EN, Maastricht, the Netherlands
[b] Leiden University, Leiden Institute of Advanced Computer Science, Snellius Building, Niels Bohrweg 1, 2333 CA, Leiden, the Netherlands
[c] Flinders University, College of Science and Engineering, Tonsley Campus, 1284 South Roach, Clovelly Park, 5042, Adelaide, Australia

ARTICLE INFO

Article history:
Received 8 September 2022
Received in revised form 2 May 2023
Accepted 4 May 2023
Available online 9 May 2023

Dataset link: https://github.com/Ludeme/Ludii

Keywords:
General game playing
Pattern matching
AI and games
Ordering propositions

ABSTRACT

In many board games and other abstract games, patterns have been used as features that can guide automated game-playing agents. Such patterns or features often represent particular configurations of pieces, empty positions, etc., which may be relevant for a game's strategies. Their use has been particularly prevalent in the game of Go, but also many other games used as benchmarks for AI research. In this paper, we formulate a design and efficient implementation of spatial state-action features for general games. These are patterns that can be trained to incentivise or disincentivise actions based on whether or not they match variables of the state in a local area around action variables. We provide extensive details on several design and implementation choices, with a primary focus on achieving a high degree of generality to support a wide variety of different games using different board geometries or other graphs. Secondly, we propose an efficient approach for evaluating active features for any given set of features. In this approach, we take inspiration from heuristics used in problems such as SAT to optimise the order in which parts of patterns are matched and prune unnecessary evaluations. This approach is defined for a highly general and abstract description of the problem—phrased as optimising the order in which propositions of formulas in disjunctive normal form are evaluated—and may therefore also be of interest to other types of problems than board games. An empirical evaluation on 33 distinct games in the Ludii general game system demonstrates the efficiency of this approach in comparison to a naive baseline, as well as a baseline based on prefix trees, and demonstrates that the additional efficiency significantly improves the playing strength of agents using the features to guide search.

1. Introduction

In research on machine learning for automated game-playing agents and Artificial intelligence (AI), the focus is often on achieving state-of-the-art or superhuman performance in one or a handful of games. In recent years, this is often based on combinations of Monte-Carlo Tree Search (MCTS) [50,32,14] and deep neural networks (DNNs) [53]. In principle this combination of techniques can be successfully applied [88,2,90,58,89,105,67,112,25,24] to a wide variety of games. However, in practice the high computational requirements make it infeasible to scale this up to large-scale studies that involve training

* Corresponding author.
E-mail address: dennis.soemers@maastrichtuniversity.nl (D.J.N.J. Soemers).

https://doi.org/10.1016/j.artint.2023.103937

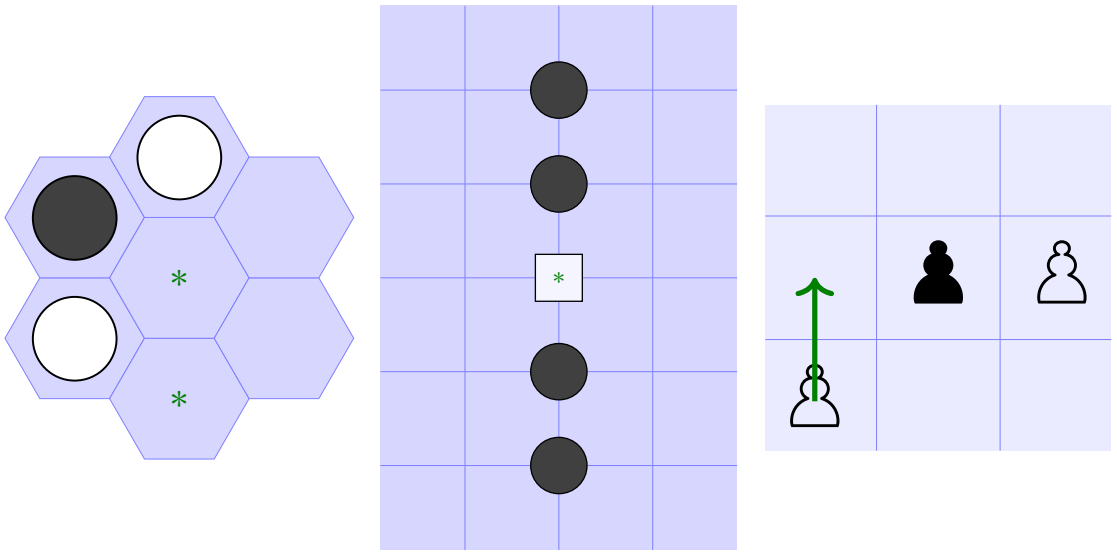


Fig. 1. Three examples of local spatial features that may be useful in various games. The leftmost feature matches actions that either complete or break a "bridge" of white pieces, depending on which player is the player to move. The middle feature matches actions that either complete or break a line of five black pieces. The rightmost feature matches actions that move the bottom-left white pawn in such a way that the black pawn becomes squashed between two white pawns.

agents for hundreds or thousands of distinct games [98], in addition to possibly many more variants of games generated automatically as possible reconstructions of games with incomplete rules [12,16].

Another common approach for improving the playing strength of search algorithms is to use higher-level features—as opposed to raw game state inputs—which may be expected to be capable of providing useful guidance to search algorithms without first being processed into useful representations through multiple layers of computation as in DNNs. Such features can be used by simpler, more efficient function approximators (e.g., linear functions, shallow neural networks, etc.) [38,55,20,111,31,41,101,45,58], or used for other purposes such as move ordering, directly playing winning moves, directly pruning losing moves, (de)prioritising other special types of moves, etc. [100,69,23,7,8,99,42,3,81,92]—without otherwise using them as inputs for parameterised functions such as policy, value, or heuristic score functions. AlphaGo also used such high-level patterns for a computationally efficient rollout policy [88]. While there are some exceptions, the majority of these approaches use local, *spatial patterns* as high-level features. These are features that describe a specific arrangement of elements (e.g. pieces, off-board indicators, empty-position indicators, etc.) that must be present (or absent) in a specific area of a game board (e.g., a small 3×3 patch of a larger Go board) for the feature to be considered active in that area. The general concept of such spatial features appears to be useful in various different games, but in each of the publications listed above they are designed, implemented, and evaluated for a single, specific game (most commonly Go)—often leveraging game-specific domain knowledge for efficient designs and implementations. In addition to possibly improving the playing strength in many games, such spatial features or patterns are attractive because they may contribute towards *human-like* game-playing [55,61], may facilitate generalisation and transfer thanks to the game-independent spatial semantics, and may be easy to visualise, interpret, and explain [15].

In this paper, we focus on the design and efficient implementation of spatial features for *general games* implemented in the Ludii general game system [17,77]. Fig. 1 depicts several examples of what such features may look like. The aim is not to achieve superhuman levels of performance as may be expected from DNN-based approaches, but rather to learn at least meaningful strategies that can substantially improve the playing strength of standard (unbiased) MCTS agents across many games, with a level of efficiency such that scaling up to many hundreds of games is feasible. To this end, we have two core contributions:

1. We provide a significant extension of previous work in which we proposed a design of such spatial features [15].
2. We explore how to implement them in a way such that the activity of features can be efficiently evaluated and used in, for instance, rollouts as used by MCTS. The algorithm we propose for this is defined for a highly general and abstract formulation of the problem, where we simply aim to optimise the order in which to evaluate propositions that appear in formulas in disjunctive normal form. This may also make it applicable to other problem domains than games (which we discuss in Section 7).

Previous work with earlier, less extensive and optimised designs of such features has already demonstrated the ability to train effective policies for a wide variety of games using such features [95,96].

Section 2 provides background information on Ludii and general game playing, as well as a brief summary of MCTS. Related work is discussed in Section 3. We formalise the design and format of our spatial features in Section 4, which also includes discussions of how relevant (spatial) aspects of states and actions are represented in the Ludii general game system. In Section 5 we propose our approach for efficiently evaluating which spatial features out of a feature set are active for any given state-action pair in the Ludii general game system. Section 6 describes and discusses the setup and results of our experiments. A discussion of potential applications beyond games is provided by Section 7. Finally, Section 8 concludes the paper and explores ideas for future work.

2. Background

In this section, we provide background information on some of the basic concepts that we build on in the remainder of the paper. We start with a discussion of the Ludii general game system [77,17], and how it relates to the research field of General Game Playing. We use the vast library of games available in Ludii for experiments, and also rely on some domain knowledge of its game, state, and move representations [76] to implement spatial features. Afterwards, we briefly describe Monte-Carlo Tree Search.

2.1. Ludii and general game playing

General Game Playing (GGP) is a field of research in which the goal is to develop agents that can play *general games* without human intervention [79,103], i.e. agents that can play any arbitrary game without requiring any game-specific domain knowledge. Such agents typically expect any game they are tasked with playing to have been implemented in a specific Game Description Language (GDL)—like the Stanford GDL [59] or the game description language of Ludii [77, 17]—such that the agents can interact with any game through a single, common API.

Game descriptions in the Stanford GDL describe the rules of games in low-level logic, which means that many commonly-used high-level game concepts (such as *square boards*, *hexagonal boards*, *lines of pieces*, *slide moves*, *step moves*, etc.) need to be expressed from scratch in low-level logic in any new game description that requires them. Game descriptions in Ludii's GDL are not written in low-level logic, but instead use a significantly larger library of *ludemes* (i.e., keywords), most of which summarise such commonly-used, high-level concepts in a single word or phrase. This design is intended to make it significantly easier—for humans, but possibly also evolutionary algorithms [19]—to write, read, and understand new game descriptions—in particular for games that are similar to "real-world" board games and primarily use rules and equipment that are already implemented as first-class citizen, and encapsulated by appropriately-named keywords, in Ludii's GDL. Ludii's object-oriented game and state representations [77] similarly make commonly-used concepts such as *game boards*, *cells*, *vertices*, *edges*, *orthogonal* and *diagonal connections*, *neighbours*, etc. readily available for any game to any agent, which allows for reliable implementations of spatial features, which rely on knowledge of such concepts.

Note that a significant amount of GGP research has been inspired by the International General Game Playing (IGGP) competition [43]. In this competition, agents only gain access to the game descriptions of any games they play when the competition starts, leaving relatively little time for any offline training. In this paper, we consider perhaps a less strict idea of GGP. In principle, we do not mind "telling" an agent in advance which games it will be expected to play, and making use of more offline training time. However, the sheer scale of working with many hundreds or thousands of games [12,98] does put practical limitations on how much training time and hardware can be used for any individual game, and makes it infeasible to program game-specific agents. This means that we cannot, for instance, handcraft strong heuristic evaluation functions or features for every game—not necessarily because we do not know which games we wish to play, but because we simply wish to play too many different games for the programming effort to be feasible. On the other hand, we can for instance leverage the knowledge that the vast majority of games we are interested in are "real-world" board games. Most of these involve spatial semantics; a game board with defined positions and connectivity relations between those positions, pieces placed on such positions, movement rules or victory conditions based on the connectivity structure, etc. This knowledge informs our choice to focus on *spatial* features.

2.2. Monte-Carlo tree search

Monte-Carlo Tree Search (MCTS) [50,32,14] is a commonly-used tree search algorithm in GGP [39,103], as well as several recent state-of-the-art game-specific agents [89,25]. It gradually builds up a search tree in an asymmetric manner, such that it focuses a greater amount of search effort on parts of the search tree that appear to be promising so far—based on intermediate estimates during the search—and less effort on less promising parts of the tree. This works by iterating through a sequence of four phases, referred to as *Selection*, *Expansion*, *Simulation* (or *Play-out*), and *Backpropagation*, for as many iterations as allowed by some search budget. This is depicted in Fig. 2.

1. *Selection*: In the Selection phase, the algorithm traverses from the root of the tree to a part of the search tree that warrants more search effort. This is typically based on a policy that provides a balance between *exploitation* of parts of the search tree that appear promising so far, and *exploration* of parts of the search tree that have received relatively little attention, such as the UCB1 policy [4].

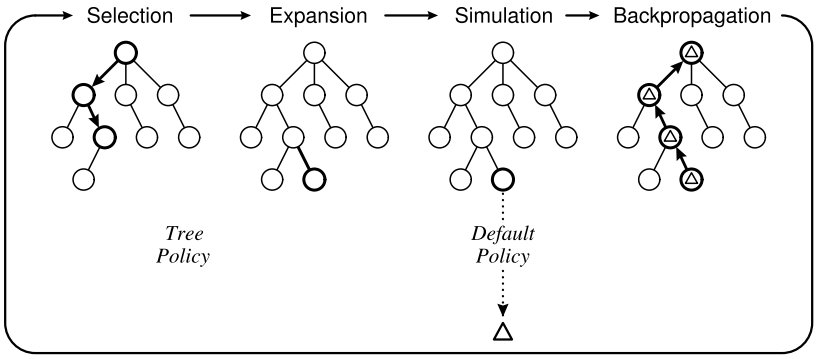


Fig. 2. The four phases of an MCTS iteration. Source of image: [14].

2. *Expansion*: In the Expansion phase, the algorithm can choose to grow the search tree. Typically this is done by adding a single new node to represent a successor—which was previously not represented by any node—of the final node that the Selection phase ended up in, but it is also possible to add multiple different new nodes at once, or add no new nodes (for instance when the Selection phase already traversed all the way to a node that represents a terminal game state without successors).
3. *Simulation*: The goal of the simulation phase is to obtain an estimate of the value of a node (or trajectory of actions) chosen by the Selection (plus Expansion) phase. The most straightforward approach for this is to run one or more random play-outs, where actions are selected uniformly at random, until a terminal game state or other limit on the play-out duration is reached, using the expanded node as starting point. This typically results in a very noisy value estimate, but it is easy to implement, efficient to run, and does not require any domain knowledge or offline learning. Alternatives include running non-uniformly random play-outs based on online [39] or offline [88] learning, or replacing play-outs altogether by a trained value function estimator [90].
4. *Backpropagation*: The Backpropagation phase propagates the obtained value estimate back through the path of the tree that was traversed in this iteration. Typically, this results in the value of a node being estimated as the average of all value estimates that have been backpropagated through that node, over all iterations that traversed that node.

3. Related work

Previous work in automated discovery of useful features or heuristics for GGP [52,27,85,39,64,49,109,65,66,110] primarily focuses on approaches that are specific to games described in the Stanford GDL [59], with its logic-based state representation. While some of these approaches may end up learning features that could be interpreted as being similar to the spatial features we consider, actually recognising and interpreting them as such first requires a human to manually analyse and interpret how the abstract, logic-based expressions in a specific game's description file relate to human-understandable concepts. Furthermore, in Ludii, many such approaches are unnecessary because spatial patterns and features can be directly implemented based on the data that is explicitly available in its game, state, and move representations [76]. Higher-level, game-wide features have also been proposed for Ludii [78], but in this paper we focus on features that can provide information about individual game states (or individual actions within game states).

Outside of GGP, spatial patterns have frequently been used in game-specific programs for games such as Chess [9,5,55], Othello [20], Breakthrough [92,82,58], Hex [45], and perhaps most commonly Go [69,23,70,111,7,8,99,42,31,91,41,3]. In all of this related work, patterns are formalised and implemented for a specific game, which means that the same formalisations are not directly applicable to general games. For example, in the game of Go, there are only two players (black and white), each of which only has a single piece type (a stone), and stones are only ever placed on the intersections of a square tiling of square cells. Therefore, in related work on patterns in Go, it is customary to only have patterns that test for particular arrangements of empty, white, and black positions—sometimes extended with board edge detectors and wildcards (positions for which it does not matter whether or not, and by what, they are occupied). For applicability to general games, our formalisation of patterns requires support for different board structures, more diverse sets of piece types and numbers of players, etc.

In the majority of research based on deep learning for board games in more recent years, *convolutional neural networks* (CNNs) [54] are used [88,90,2,58,89,105,67,112,29,25,30,94,93]. These typically operate on raw, low-level state inputs, as opposed to the higher-level features considered in this paper. However, due to the way in which CNNs implement translation invariance and weight sharing by "sliding" learned filters over the spatial dimensions of a state input, such learned filters may intuitively be understood as encoding "fuzzy" versions of spatial patterns. Note that most of this related work only uses convolutional layers at the "start" of a neural network, but follows these up with at least one fully-connected layer. This means that there is no translation invariance or other use of spatial semantics in the action-based policy outputs. This is in contrast to the spatial state-action features considered in this paper, which specifically look at a local area around an action in any given state. Fully-convolutional architectures [87], which have been demonstrated to facilitate transfer learning

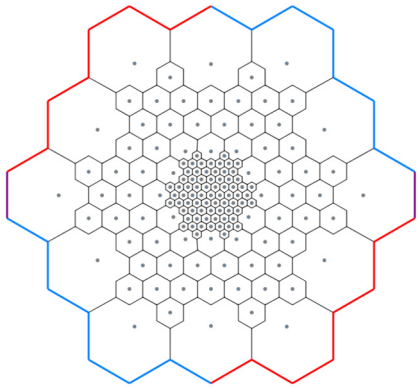


Fig. 3. The game of Fractal in Ludii. The dots mark playable sites; these are cells of a variety of shapes.

between games with different board sizes [94], bear a closer resemblance in this respect. It should also be noted that CNNs normally work best on regular tilings, such as grids of pixels or square cells, or regular tilings containing only hexagonal cells, etc. In this paper, we aim to formalise features independent of any single particular regular tiling, and even aim for them to be applicable to game boards that do not have a regular tiling (see Fig. 3). Such levels of generalisation can be present in more general (graph-based) architectures than CNNs [10], which have for instance been applied to the game of Risk [22].

4. Formalisation of spatial state-action features

In this section, we formalise our design of spatial state-action features. These are binary features $\phi(s, a)$ of game states s and actions[1] a. The basic idea is that a feature tests for one or several conditions in the neighbourhood around the positions involved in an action, and is either active ($\phi(s, a) = 1$) or inactive ($\phi(s, a) = 0$) if the conditions are satisfied or not satisfied, respectively, for s and a. Simple examples of conditions include testing whether there is a friendly piece, or enemy piece, or empty position, etc., in some position relative to (e.g., adjacent to) the position that is the destination of an action a in a state s. Before describing the design of these features in further detail, we discuss some preliminaries concerning the game, state, and action representations in Ludii [76].

4.1. Ludii game, state, and action representations

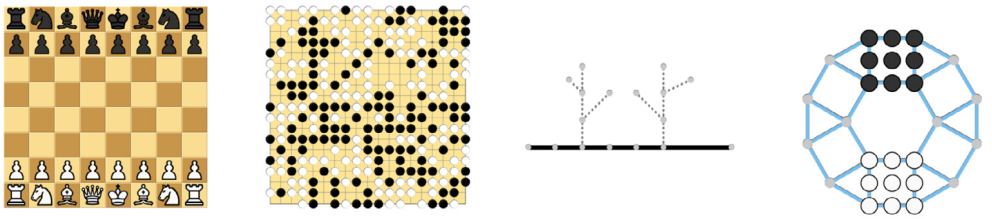
Every game's representation in Ludii includes one or more *containers*, each of which defines a "playable area" as a graph, which has vertices, edges, and—in many cases—cells (or faces). Typically there is one primary container (often simply representing a game's board), and sometimes one or more additional containers to represent supplementary areas. For instance, the game of Shogi as modelled in Ludii has additional containers to represent "players' hands", which hold captured pieces. Among all games (over 1,000) implemented in Ludii at the time of this writing, there is not a single game with a meaningful connectivity structure or spatial semantics within any container other than a game's primary board. Hence, in this paper, we only consider using spatial features within any game's main board. While the graphs used for many games define vertices, edges, and cells, the vast majority of games only use one of those three types in their rules. For example, chess only uses cells, Go only uses vertices, and the implementation of Hackenbush in Ludii only uses edges (see Fig. 4). A few games use a combination of multiple types, such as Triple Tangle, which uses all three types.

4.1.1. Ludii board geometry

Given any graph as described above, the *connectivity* relations between certain elements of the graphs are typically crucial for the game's rules and its strategies. For example, movement rules of pieces in Chess are typically defined in terms of the connectivity relations between cells of the board, win conditions in line-completion games such as Tic-Tac-Toe or connection games such as Hex use similar relations to determine whether or not specific collections of cells count as lines or connections, etc. Fig. 5 provides several examples of orthogonal and diagonal connections between cells, and between vertices, on square grids.

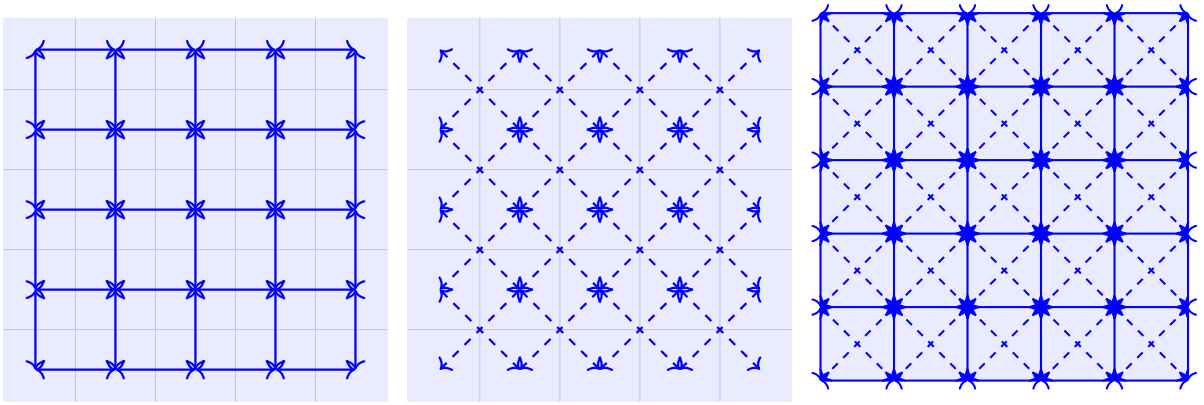
For the features discussed in this paper, we only focus on orthogonal connections. Whether or not any pair of cells or vertices are considered to be orthogonally connected in Ludii generally closely matches human intuition, in particular on graphs constructed from regular tilings (see Fig. 5). We provide more formal definitions as follows:

[1] Throughout this paper, we follow standard reinforcement learning terminology [102], referring to the decisions of agents (or players) as actions. Within the Ludii general game system, there is a distinction between actions and moves, with actions describing primitive modifications of game states, and moves—formed by sequences of one or more actions—describing decisions made by players. In this paper, there is no need for any such distinction.



(a) Chess. (b) Go. (c) Hackenbush. (d) Triple Tangle.

Fig. 4. Four different games in Ludii that use different graph element types in their rules. Chess only involves the cells of its board. Go only involves the vertices of its board. Hackenbush only involves the edges of its graph. In Triple Tangle, pieces are placed on cells, vertices, and edges.



(a) Orthogonal connections between cells. (b) Diagonal connections between cells. (c) Orthogonal and diagonal connections between vertices.

Fig. 5. Various types of connections on square grids.

- Two cells c_1 and c_2 ($c_1 \neq c_2$) are orthogonally connected if and only if they share at least one edge.
- Two vertices v_1 and v_2 ($v_1 \neq v_2$) are orthogonally connected if and only if they share at least one edge.

We do not build in explicit support for other types of connections [13], such as diagonal connections, in our features. This is primarily to avoid the large increase in the space of features that could be represented. Note that in many cases, such as square tilings, diagonal connections may still be expressed indirectly as sequences of two orthogonal connections. Since we are, in most cases, only interested in a single type of graph element (only cells in Chess, only vertices in Go, etc.), we will frequently use the term "*site*" to refer to one of these elements—regardless of its actual type.

In many games, rules are not only based on the connectivity structures between sites, but also on a sense of "continuity" of connections. For instance, a queen in Chess may follow a sequence of many connections in a single move, but only if they all continue in the same "direction." Ludii supports this by automatically computing *radials* [13], which may intuitively be understood as sequences of consecutive steps such that they continue in the same direction. More formally, let s_1 and s_2 ($s_1 \neq s_2$) denote a pair of sites, such that there is an orthogonal step from s_1 to s_2. Let s_3 ($s_3 \neq s_2$) denote whichever connected site of s_2 has the smallest absolute change in angle between an arrow pointing from s_1 to s_2, and an arrow pointing from s_2 to s_3. Let θ denote this absolute change in angles. If $\theta < 0.25$ rad, we include the step from s_2 to s_3 after the step from s_1 to s_2 in a single radial (this process may be repeated many times for a longer radial). If $\theta \geq 0.25$ rad, the radial does not continue. Note that, for the computation of these angles, we assume that every site has x- and y-coordinates that define its position. This is enforced in Ludii, in part also to facilitate visualising games and game states in a graphical user interface, but—especially for arbitrary graphs—these coordinates are not necessarily guaranteed to be meaningful for gameplay. In practice, since we focus on "real-world" games also played by humans, such coordinates often tend to be meaningful. Fig. 6 depicts an example of a full radial.

4.1.2. Ludii state representation

While Ludii's state representation contains many variables [76], the core variables that are important to understand for the purposes of this paper are three bit arrays referred to as `empty`, `who`, and `what`. For simplicity, we assume a game that uses only a single graph element type (only cells, only edges, or only vertices), with $P \geq 1$ different players, $M \geq 1$ different piece types, and $N \geq 1$ different sites. For any give game state s, the three bit arrays encode the following data:

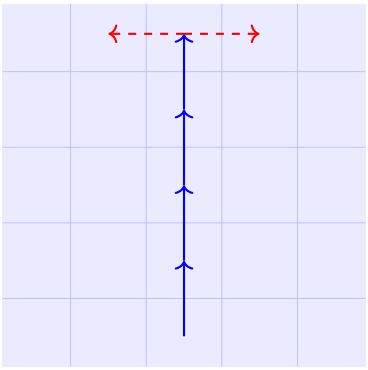


Fig. 6. Example of a radial. From the bottom to the top, a radial can be built up of consecutive orthogonal steps with absolute changes in angle less than 0.25 rad (in fact, angles of 0 rad in these cases). When the top row is reached, and there no longer exists any step in the same direction, there is a tie for the lowest absolute change in angle between two steps—one westwards and one eastwards. The absolute value of the change in angle ($\frac{1}{2}\pi$) exceeds 0.25 rad, so these are not valid continuations and the radial ends.

- `empty`: A bit array of length N with, for every site $0 \leq i < N$, a value of 1 at index i if and only if site i is empty in s.
- `who`: A bit array of length NB, representing N consecutive chunks of B bits each. Here, $B = \left\lfloor 2^{\lceil \log_2(X) \rceil} \right\rfloor$, and $X = \left\lfloor \log_2(P+1) + 1 \right\rfloor$. X is the number of bits required to encode positive integers up to and including $P+1$, and B is the lowest power of 2 that is greater than or equal to X. Chunks are preferred to have powers of 2 as size, because that means that a single chunk will never be split up over different 64-bit `long` values in Java when the bit array is implemented as an array of `long` values, which in turn is important for the implementation of efficient bitwise operators to extract or modify chunk values. For every site $0 \leq i < N$, the bits in the N^{th} chunk are set such that they form the binary representation of the integer indicating the player that occupies site i, where a value of 0 indicates a neutral piece or no occupier (empty site), and a value of $P+1$ indicates occupied by a "shared" piece.
- `what`: A bit array of length NB, with $X = \left\lfloor \log_2(M) + 1 \right\rfloor$ and B defined as above, such that it has N consecutive chunks of B bits each, where the chunks can encode positive integers up to and including M. For every site $0 \leq i < N$, the bits in the N^{th} chunk are set such that they form the binary representation of the integer indicating the piece type that occupies site i, where a value of 0 indicates no piece (i.e., an empty site).

4.1.3. Ludii action representation

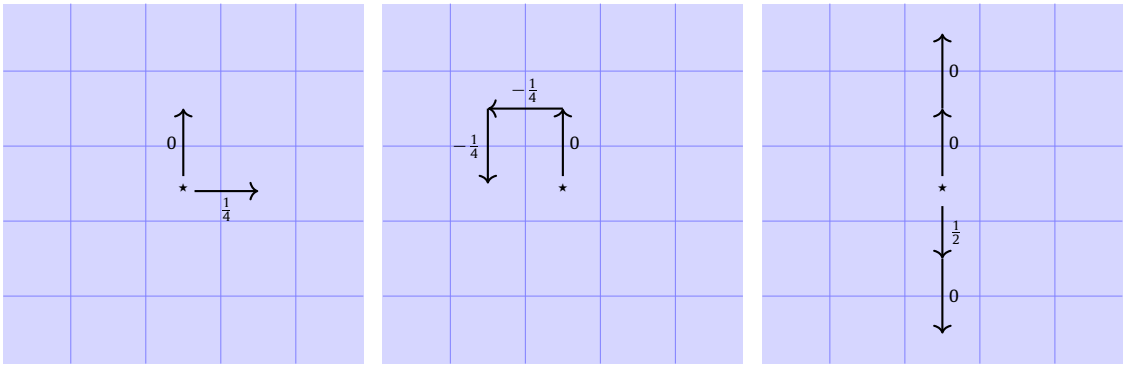
Like the state representations, action representations in Ludii can contain many variables [76]. The two variables that are most important for the purposes considered in this paper are `from` and `to`. These are typically sites (i.e., integers $0 \leq i < N$ in games with N sites), which may intuitively be understood as the "source" (position that a piece moves away from) and "destination" (position that a piece moves towards or is placed on), respectively. Sometimes `from` and `to` have the same value (for instance in games such as Go, Hex, or Tic-Tac-Toe), and sometimes they have values of -1 (for instance for moves where players opt to pass). It is relatively uncommon, but there can be games where multiple distinct legal actions in a single state have identical `from` and `to` values. Such actions are *aliased* (i.e., impossible to distinguish) when using a feature space that only accounts for these two variables.

4.2. Representing relative positions in spatial features

The basic premise of the spatial features $\phi(s, a)$ that we propose is that they should encode patterns, or arrangements, where certain elements (empty positions, stones, pawns, queens, etc.) are present or absent in locations relative to some particular point of interest—such as a `from` or `to` position of an action a. When dealing with a specific game such as Chess or Go, as in much of the related work discussed in Section 3, with a fixed and regular tiling of sites on a board, it is straightforward to define relative positions based on offsets that are applicable to any position on such a board.

4.2.1. Describing relative positions as walks

For general games, we propose to define relative positions as *walks* of 0 or more steps along orthogonal connections, where every step is represented by a real number $-1 \leq \rho \leq 1$ that describes the ratio of a 360° clockwise turn that should be taken, relative to a "current" direction, before taking the next step. To make this easier to understand, Fig. 7 provides several examples of different walks on a grid of square cells. Cases where we know in advance that we are dealing with a regular tiling of sites, which all have the same number of connections, are still the easiest cases to work with. For example, all example walks in Fig. 7 use rotations that are multiples of $\frac{1}{4}$, since these are the most natural rotations to use given four-sided cells that have four orthogonal connections each. Other rotations may be more natural on other boards, such as multiples of $\frac{1}{6}$ for tilings of hexagonal cells (see Fig. 8a). Note that constraining rotations ρ to a smaller range of values,

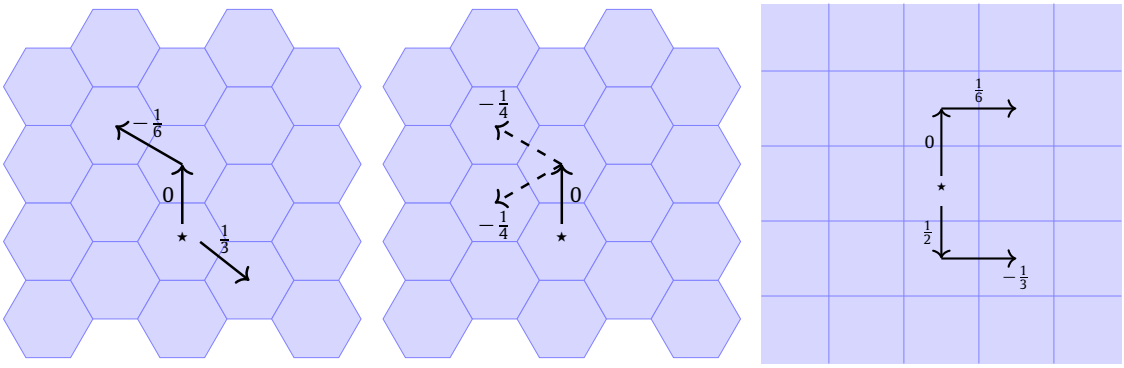


(a) Two example walks, each consisting of a single step.

(b) A single example walk, consisting of three steps.

(c) Two example walks, each consisting of two steps.

Fig. 7. Several example walks, describing sites relative to some starting point marked by $\star$, on regular tilings of sites with four orthogonal connections each (i.e., square cells). (a) A step with $\rho = 0$ travels northwards, whereas a step with $\rho = \frac{1}{4}$ travels eastwards, due to a $360° \times \frac{1}{4} = 90°$ rotation clockwise from the "default" direction. (b) Rotations are relative to the "current" direction. Hence, after the second step of this walk with a rotation $\rho = -\frac{1}{4}$ already resulted in a westwards facing, an additional rotation of $-\frac{1}{4}$ results in a southwards step. (c) A two-step $\{\frac{1}{2}, 0\}$ walk may be viewed as a vertically reflected, or a rotated (by 180°) version of a two-step $\{0, 0\}$ walk.



(a) Two example walks on a grid of hexagonal cells.

(b) Example walk with two possible destinations from $\star$.

(c) Two example walks resolved on square cells.

Fig. 8. Several more complex example walks. (a) Because hexagonal cells have six orthogonal connections each, rotations are most naturally expressed as multiples of $\frac{1}{6}$. (b) If a walk has a step with a rotation of $-\frac{1}{4}$ on a grid of hexagonal cells, this can be interpreted as either $-\frac{1}{6}$ or $-\frac{1}{3}$, because it is perfectly in the middle of those two values. Hence, a single walk can "split" into two different walks (depicted with dashed arrows), and it can represent either of the two destinations as position relative to $\star$. (c) On a grid of square cells, rotations of $(-)\frac{1}{6}$ and $(-)\frac{1}{3}$ both resolve to the same closest "natural" rotation of $(-)\frac{1}{4}$.

such as $\rho \in [0, 1)$, would still be sufficient and equally representative. However, we sometimes find the larger $[-1, 1]$ range to be more convenient and easier to use when manually reading or writing patterns for testing or development purposes.

Rotations other than these "most natural" rotations can still be used, simply by rounding them to the closest natural rotation for any given site. If a rotation is exactly[2] in the middle of two consecutive natural rotations for a given site, we "split up" the walk into two different walks; one for each of the rotations we can round to. This behaviour is illustrated in Fig. 8. This functionality can be useful for transfer between games with different board geometries, and also for games played on boards that are not regular tilings (such as Fractal; see Fig. 3). This representation is intended to preserve spatial semantics as closely as possible even when resolving walks in situations where some of the described rotations do not match the natural rotations of the sites involved. For example, rotations of 0 and $\frac{1}{2}$ will always be "opposites" of each other, and a rotation of $\frac{1}{4}$ (relative to a default northwards facing) will always face approximately eastwards—regardless of whether it is evaluated on a site that has 4 orthogonal connections, or 400 or any other number of orthogonal connections.

4.2.2. Representing off-board space

Many game's rules do not solely involve positions and connectivity relations defined as an arbitrary graph, but also incorporate a sense of "direction." As described in Subsubsection 4.1.1, Ludii automatically computes *radials*, which represent

[2] In practice we use a tolerance value of $\epsilon = 0.02$.

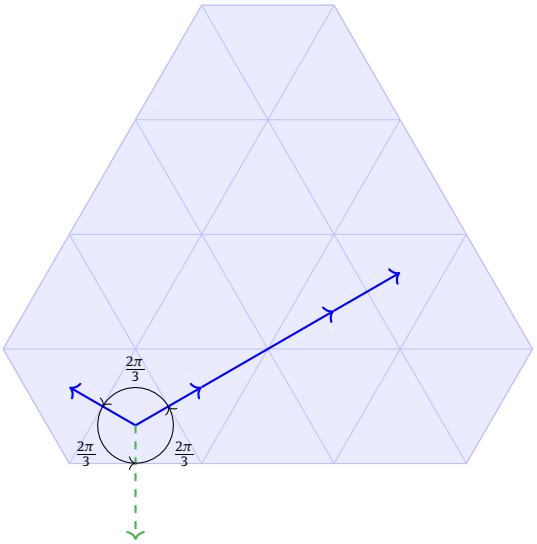


Fig. 9. The solid blue arrows represent two radials that each start with an orthogonal step, originating from the same cell. There is an angle of $\frac{2\pi}{3}$ rad between them. We insert an artificial "off-board" connection, represented by the green dashed arrow, to complete the circle. (For interpretation of the colours in the figure(s), the reader is referred to the web version of this article.)

sequences of steps that follow along a single direction. However, as depicted in Fig. 6, these radials stop when the border of a game board is reached. This also leads to the issue that, for example, all the sites along the border of a square tiling only have three orthogonal connections (or two for the corners), whereas inner sites have four connections. Arguably a more natural representation would introduce additional connections leading to "off-board" positions, such that we can detect at what point of a walk a step in a certain direction would cause us to wander "off the board." While it may be considered obvious where such additional connections would have to be added for certain specific cases, such as regular square tilings, this is not always true for more unconventional boards, or arbitrary graphs, which may also be used in Ludii. To support more general cases, we propose a series of steps to automatically compute such off-board connections, based on "extending" existing radials past their stopping point. While we cannot guarantee that this produces the expected result in any arbitrary case (or even just formally define what the expected result would be in any arbitrary case), we can demonstrate the results for a variety of common cases.

1. Completing triangles Whenever a site has only two outgoing radials that start with an orthogonal step, with an angle of $\frac{2\pi}{3}$ between them, we insert an additional off-board step at another $\frac{2\pi}{3}$ angle. This "completes" triangular cells along a board edge, as depicted in Fig. 9.

2. Continuing opposite radials Let s_1 denote a site with an orthogonal step to another site s_2, such that s_2 has a radial back to s_1 that does not extend beyond s_1. In such a case, we insert an artificial off-board connection from s_1, in the same direction that the step from s_2 to s_1 points to. This handles common cases such as sites along the edge of a chessboard or Go board, as depicted in Fig. 10. This step is skipped if the previous step (for completing triangles) already introduced new connections.

3. Encouraging uniform angles While it is possible to play games on arbitrary graphs with arbitrary structures, the vast majority of games use regular or semiregular tilings with regular polygons. Hence, as a heuristic, we prefer uniform angles between all outgoing radials starting with orthogonal steps for any single given site. For any site to which some artificial connections were already added in the previous step, we introduce additional ones if this can ensure that the angles between any pair of adjacent connections become equal (not allowing the total number of connections to be more than doubled). Fig. 11 depicts, as an example, how this affects two of the corners of a rhombus-shaped board of hexagonal cells, as used in the game of Hex.

Walks into off-board space Using the additional off-board connections as described above, we can detect when a walk wanders off the board. However, beyond those steps, the "off-board" space is not further modelled with additional sites and connections. This means that, once a walk wanders off the board, it can never return; the final destination of such a walk is always assumed to be an "off-board" position.

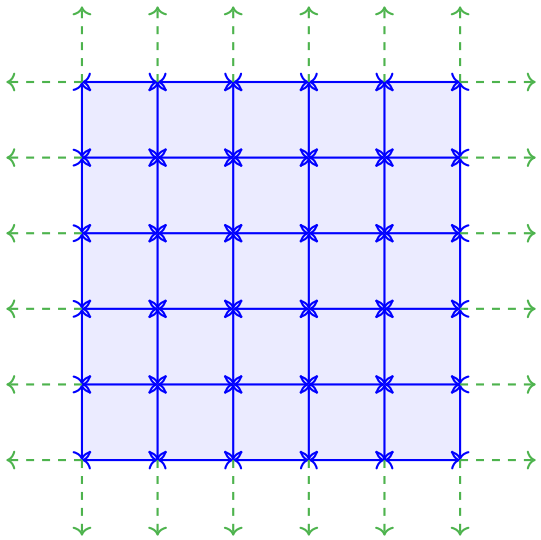


Fig. 10. The solid blue arrows represent radials for orthogonal steps on a 6×6 Go board (assuming play on the intersections). The dashed green arrows represent off-board continuations of these radials.

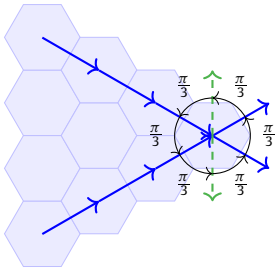


Fig. 11. The solid blue arrows represent radials, plus two off-board continuation steps, for the rightmost cell. The dashed green arrows represent two additional off-board connections added to ensure uniform angles between adjacent connections.

4.3. Representing actions and state elements in spatial features

A spatial state-action feature $\phi(s, a)$ must encode relevant (spatial) aspects of the action a, as well as some elements that must be present or absent in the game state s, typically in a local area around the sites affected by a, for the feature to be considered active. For every such feature, we leave one variable site referred to as the *anchor*, denoted $\star$. Any other aspects are defined relative to $\star$ using walks, as explained in Subsection 4.2. Firstly, we define the following four action-related properties, for which any feature $\phi(s, a)$ can specify that they must be present in positions i relative to an anchor $\star$ for $\phi(s, a)$ to be considered active:

- `to`: Requires that the `to` position of action a coincides with the position i relative to $\star$.
- `from`: Requires that the `from` position of action a coincides with the position i relative to $\star$.
- `last to`: Requires that the `to` position of the last action a', which led to s, coincides with the position i relative to $\star$.
- `last from`: Requires that the `from` position of the last action a', which led to s, coincides with the position i relative to $\star$.

Any state-action feature $\phi(s, a)$ must have at least a specifier for either `to` or `from`—because otherwise it would simply be a state feature $\phi(s)$ that could not distinguish between any actions—and it can have specifiers for both. If a feature has specifiers for either `last to` or `last from` (or both), we refer to it as a *reactive* feature. Reactive features are generally more efficient to use because they only need to be evaluated in a local area around the last action, which they "react" to.

Secondly, we define several state properties, for which any feature $\phi(s, a)$ can specify that they must be present or absent in positions i relative to an anchor $\star$ for $\phi(s, a)$ to be considered active. Note that for each of these, we consider affirmative as well as negated variants:

- `empty`: Requires that the position i relative to $\star$ is (not) empty, i.e. the i^{th} bit in the `empty` bit array of s must (not) be set to 1.

- `friend`: Requires that the position i relative to $\star$ is (not) occupied by a friendly piece, i.e. the bits in the i^{th} chunk of the `who` bit array of s must (not) represent the integer value p, where p is the player to move.
- `enemy`: Requires that the position i relative to $\star$ is (not) occupied by an enemy piece, i.e. the bits in the i^{th} chunk of the `who` bit array of s must (not) represent an integer value other than 0 or p, where p is the player to move.
- `off`: Requires that the walk specifying i, relative to $\star$, leads to a position that is (not) "off the board" (see Subsubsection 4.2.2).
- `item`: Requires that the position i relative to $\star$ is (not) occupied by a specific piece type with a specified index k, i.e. the bits in the i^{th} chunk of the `what` bit array of s must (not) represent the integer value k.
- `connectivity`: Requires that the position i relative to $\star$ is (not) a site that has a specified number k of orthogonal connections.
- `region proximity`: Requires that the position i relative to $\star$ is (not) closer than $\star$ to the *region* defined in the game with a specified index k. Regions are predefined sets of sites that typically have a specific, important meaning in a game's rules (e.g., a region of sites that a player must reach or to win).

Note that these state and action properties are not necessarily sufficient to perfectly distinguish all unique states and actions in all games; in some games, there may be pairs of distinct states s and s', or distinct actions a and a' that will always be indistinguishable based on only these properties. The choice to include these properties in the representation and no others is a subjective choice. This is primarily based on our intuition of which properties permit efficient evaluations of features, are relevant and important to include in some games, and also generally applicable to the extent that they are not only relevant in a small, highly specific selection of games. For most of these properties, analogous properties are also frequently included in related work on game-specific patterns (see Section 3). Similarly, approaches based on deep learning with convolutional neural networks frequently use channels encoding similar data, such as binary channels indicating presence or absence per piece type [89,25,93]. An example of a more advanced property that we considered is a "line-of-sight" property, which tests whether or not a position is in line of sight—without other obstructions in between—of a particular player or piece type. Including features with such properties was found to require an excessively large amount of memory in preliminary testing.

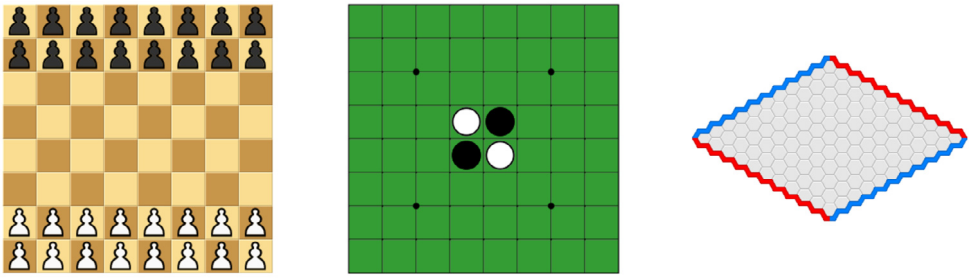
4.4. Exploiting symmetries in features

A common approach to improve sample efficiency and training speed in AI for games, as well as machine learning more generally, is to exploit various symmetries through weight sharing. For example, a significant amount of related work on patterns in games such as Go uses translational weight sharing; the same weight is associated with a pattern, regardless of where it appears on the game board. Other forms of symmetry that are frequently leveraged include reflection, rotation, and player colour inversion [99,42,41,88,90,58]. Outside of game AI applications, CNNs [54]—which are ubiquitous in deep learning approaches for machine learning problems with image-based inputs—implement weight-sharing through translation equivariance. Other architectures have been developed to exploit additional symmetries in various domains [51,10,28].

Many of these symmetries are not necessarily perfectly applicable to all games. Consider, for example, a simple feature that tests for moves that capture a queen in chess. Capturing the opponent's queen may be more or less valuable depending on which position it is located in, which suggests that such a feature should have different weights for different anchor positions. On the other hand, it is likely that—even if the exact position may be relevant—capturing the opponent's queen will in general be likely to be a valuable move, which suggests that training can be sped up significantly by sharing the same weight for such a feature regardless of the exact position of the anchor when it is active.

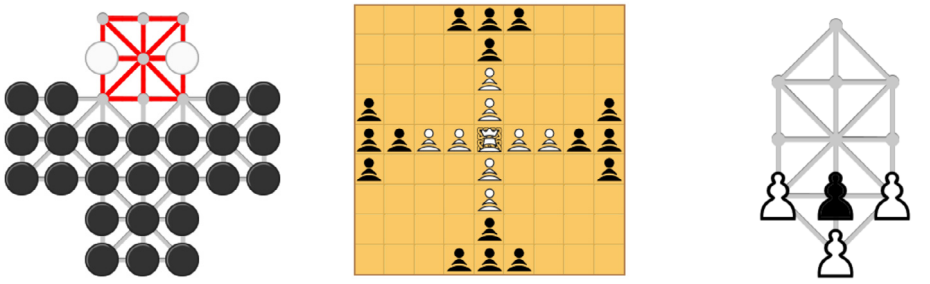
In this work, we share the same weight across all translations (distinct anchor positions), rotations, and reflections per feature. This improves generalisation and computational efficiency, possibly at a cost in representational capacity. We discuss three examples of well-known games where we know some of these symmetries to be "incorrect", and ways in which negative effects on the representational capacity of policies based on our features are mitigated in these games:

1. In the game of Breakthrough (see Fig. 12a), players are only allowed to moves their pawns "forwards" (diagonally or orthogonally), towards the board edge opposite their starting positions. This suggests that patterns that are representative of strong or weak situations likely no longer represent the same after rotation or vertical reflection—except if the player colours also were to be inverted. However, our state-action features $\phi(s,a)$ also encode a representation of the action a, and they need only be evaluated for legal actions a. Because such rotations or vertical reflections would only be applicable to illegal actions, this form of weight-sharing is harmless (as well as useless) in this situation.
2. In the game of Reversi (see Fig. 12b), it is well known that the corners of the board are particularly important to control. This suggests that some patterns may be more or less important depending on whether or not they are close to a corner, and translational weight sharing may not be appropriate. However, we include the ability to detect off-board positions relative to the anchor, and such elements can be used by features $\phi(s,a)$ to ensure that $\phi(s,a) = 1$ only if the anchor is in a particular offset relative to any corner—still allowing for generalisation between different corners through combinations of translation and rotation or reflection.
3. In the game of Hex (see Fig. 12c), one player aims to connect the northeast and southwest sides of the board, whereas the other player aims to connect the northwest and southeast sides of the board. This means that for each player, there



(a) Breakthrough. (b) Reversi. (c) Hex.

Fig. 12. Three examples games where certain symmetries may not be applicable.



(a) Asalto. (b) Tablut. (c) Jeu Militaire.

Fig. 13. Three examples of highly asymmetric games.

is generally one axis along which it is significantly more valuable to extend a connection than the other axis, which means that weight sharing across some rotations may be less sensible than others [45]. However, each of the pairs of sides per player is defined as a region in the game's description in Ludii. Therefore, the inclusion of elements that test for proximity to either of these regions in certain positions relative to the anchor can ensure that a feature can or cannot be active under certain rotations.

In games such as Go, it is also common to leverage symmetry under colour inversion; for example, training datasets can be augmented by generating additional sample states where all black stones are changed to white stones, white stones changed to black stones, and training labels (such as the outcomes of games) similarly inverted. We do not consider any generalisation of this form in this paper, instead opting to train sets of features and weights separately for every player's perspective. One reason for this is that we also aim to train policies for games with $n > 2$ players, for which it is not immediately clear how colour inversion should work. Another reason is that, even among two-player games, there is a significant number with a high degree of asymmetry, such as many hunt games [71], Tablut [57], and Jeu Militaire [60]; see Fig. 13. In these games, two opposing players can have different piece types, different move rules, different initial setups, different victory conditions, etc.

4.5. Examples of complete features

Fig. 14 depicts visualisations of four examples of complete features that could be modelled using the design described in this section. In the first three subfigures, a green star is used to represent both the anchor and the `to` position. In the final subfigure, a green arrow represents the `from` and `to` positions, of which one is customarily used as anchor. Internally, positions of all other elements are represented by walks relative to anchors, as described previously, but explicit visualisations of these walks are omitted to improve visual clarity. Fig. 14a is a feature that matches actions placing a piece in between two existing white pieces along an orthogonal axis. This can be used to recommend players to make a winning move (if they are the white player), or block a winning move for the opponent (if they are the black player) in Tic-Tac-Toe. Fig. 14b shows a similar feature, except for diagonal rather than orthogonal lines of pieces. Depending on the perspective, Fig. 14c can be used to incentivise either completing a bridge of white pieces, or breaking one. This is an important tactic in connection games such as Hex [11,18]. Finally, Fig. 14d recommends a diagonal movement to capture a black pawn, starting from a position that does not have a white pawn to protect it on either of the diagonal cells below it. Such a pattern is

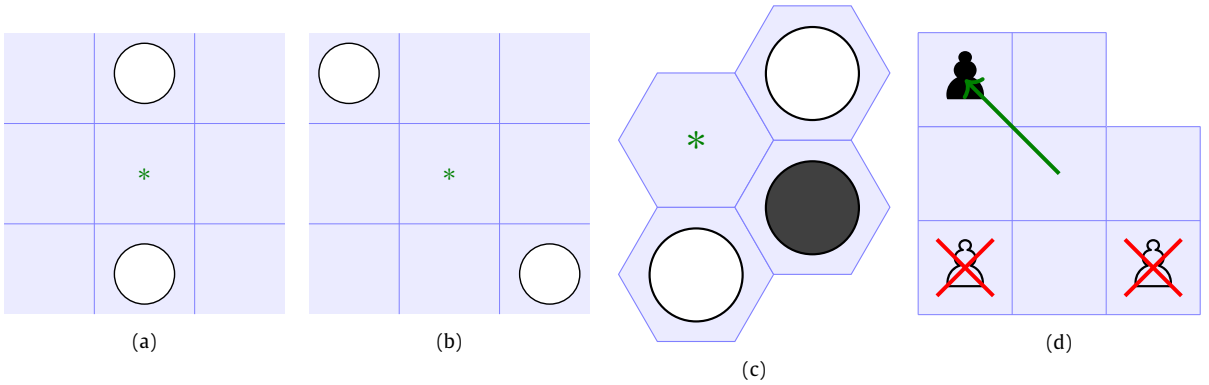


Fig. 14. Four visualisations of examples of complete state-action features.

particularly useful for games such as Breakthrough (in which pawns that capture diagonally are the only piece type), but may also have some value in games with more varied piece types like Chess.

5. Efficiently evaluating spatial state-action features

When using state-action features $\phi(s, a)$ to guide an MCTS process, it is important that the activity level of such features for any relevant state-action pair (s, a) can be computed efficiently. If this is not efficient, the computational overhead may reduce the number of iterations that can be run within any given time budget too much, which may in turn reduce the playing strength relative to an unguided, more efficient search. Efficiency is particularly important for the play-out phase, where—for the sake of efficiency—it is common to use fewer, smaller, or otherwise more efficient features or policies [42,31,88] than in the selection phase. Nevertheless, also in the selection phase it is naturally advantageous if features can be computed efficiently.

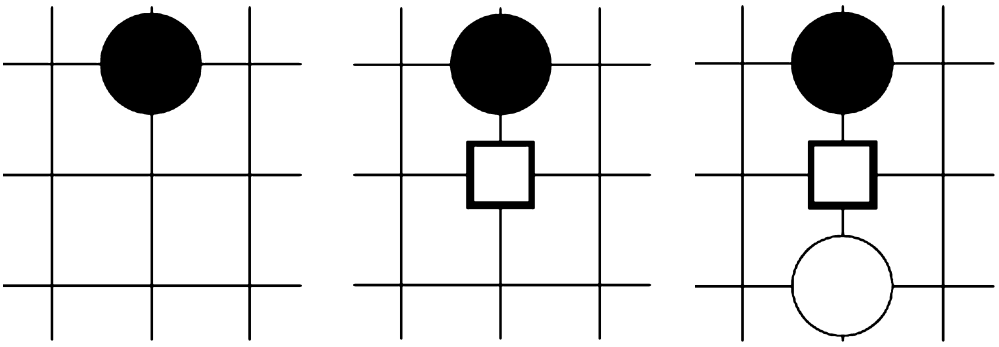
One common approach in computer Go is to directly index into a table, using variants of Zobrist hashing [113], to retrieve active patterns for any given state s and position corresponding to an action a. Such an approach only works when, for any considered pattern size, only all *complete* patterns—without any "wildcard" or "do-not-care" elements or positions—are used [40,99,91,41,3]. For example, in Go this approach can be used if all patterns of a given size, such as 3×3, fully specify every position within that area as being either empty, black, white, or off-board. With our feature formalisation, applicable to arbitrarily-shaped graphs, this is impossible to guarantee. Furthermore, for games with many more piece types than just the two of Go, such as Chess with twelve piece types (six per player), the number of unique, fully-specified patterns that could be enumerated—even for a small area such as 3×3—would be excessively large.

The second common approach for evaluating patterns in computer Go is to store patterns to be evaluated in data structures that account for generalisation relations between patterns, and avoid matching patterns that can already be inferred to not be a match based on evaluations of other patterns. Consider, for example, the three patterns for Go depicted in Fig. 15. All three patterns share the same requirement for a black stone above the centre, and the last two also share the same requirement for the centre to be empty; the first pattern *generalises* the last two, and the second also generalises the last. In any situation where the first pattern does not match, the last two need not be evaluated because they will for sure also not match. Similarly, the third pattern can be skipped if the second pattern does not match. This is typically implemented by storing patterns in tree structures [55,68,69]—such as prefix trees, where more general patterns are "prefixes" of more specific patterns—or deterministic finite state automata [107]. The approach we propose in this section bears some resemblance to these ideas, but includes additional optimisations.

5.1. Instantiating features

Let $\Phi = \{\phi_0, \phi_1, \ldots, \phi_n\}$ denote a feature set of n spatial features ϕ_i, as formalised in Section 4, for some arbitrary game. We aim to efficiently compute binary feature vectors $\boldsymbol{\phi}(s, a) = \left[\phi_0(s, a) \quad \phi_1(s, a) \quad \ldots \quad \phi_n(s, a)\right]$ for any state-action pair (s, a). For every unique possible anchor position $\star$ in the given game, we *instantiate* every feature $\phi_i \in \Phi$ by resolving all walks in ϕ_i from $\star$. After resolving the walks, we know the specific positions for which ϕ_i has requirements relative to the anchor $\star$. This process of instantiating the feature is repeated for all rotations and reflections, where $\star$ is assumed to represent the origin, and the number of orthogonal connections from $\star$ determines the number of rotations to consider. Note that feature instantiations are specific to a single player's perspective, because the meaning of "friend" or "enemy" is different for different players. For notational brevity, we leave the dependence on a specific player p implicit; there is no ambiguity because the player for whom features $\phi(s, a)$ are evaluated is always the player that can select the action a in the state s.

Recall that every feature ϕ_i must specify a walk for at least an action's `from` or `to` position, and it can specify walks for both. When we wish to compute which features are active for a state-action pair (s, a), the anchors for which any



(a) This pattern generalises b and c.

(b) This pattern is generalised by a, but generalises c.

(c) This pattern is generalised by a and b.

Fig. 15. Three example patterns for Go in 3×3 windows. Small squares denote positions that must be empty for the pattern to match, whereas any other empty positions are wildcard positions, the contents of which are irrelevant to pattern matching.

Algorithm 1 Naive evaluation of active features.

Require: Set of relevant feature instantiations $\mathcal{F}(s,a) = \{f_0, f_1, \ldots, f_k\}$.
$\boldsymbol{\phi} \leftarrow \mathbf{0}$ // Init. to zero vector.
for each feature instantiation $f_i \in \mathcal{F}(s,a)$ **do** // Test whether f_i is a match.
  `match` $\leftarrow$ `true`
  for each condition c of f_i **do**
    if (s,a) violates c **then**
      `match` $\leftarrow$ `false`
      break
    end if
  end for
  if `match` **then**
    $j \leftarrow$ feature index of which f_i is an instantiation
    $\boldsymbol{\phi}[j] \leftarrow 1$
  end if
end for
return feature vector $\boldsymbol{\phi}$

instantiations were generated are irrelevant; rather, we require the ability to find any instantiations for which any specified `from` or `to` walks match the action a. Hence, we store instantiations in maps that can be accessed quickly with hash keys generated from the player index p, as well as any mix of `to`, `from`, `last to`, and `last from` indices. Keys including either (or both) of the last two are used to allow for fast retrieval of relevant instantiations of reactive features for any state-action pair (s,a).

Let $\mathcal{F}(s,a) = \{f_0, f_1, \ldots, f_k\}$ denote such a set of feature instantiations that can be relevant to the state-action pair (s,a) according to the action-related properties. Any requirements that features may have for `off`, `connectivity`, or `region proximity` elements can already be evaluated at instantiation time; instantiations with conditions that are violated can already be removed, and for the remaining instantiations these conditions need no longer be checked at runtime. This leaves only `empty`, `friend`, `enemy`, and `item` conditions to be evaluated at runtime; these can all be implemented as simple tests involving the `empty`, `who`, and `what` bit arrays of a game state s. Algorithm 1 provides pseudocode for a simple, naive algorithm to evaluate which features are active for any given state-action pair (s,a) from such a set of instantiations. It simply evaluates all the conditions of every feature instantiation in sequence, which is straightforward to implement, but slow because it does not leverage any overlap in conditions or other relationships between multiple instantiations.

5.2. Features as disjunctions of conjunctions

As a first step towards a more efficient approach for computing feature vectors from a set of relevant instantiations $\mathcal{F}(s,a)$, we discuss how every feature (represented by one or more feature instantiations) may be represented as a disjunction of conjunctions of propositions (i.e., a logical formula in disjunctive normal form).

Let $f \in \mathcal{F}(s,a)$ denote any arbitrary feature instantiation in the set. Let $\phi(f)$ denote the original feature of which f is an instantiation. Let $\Phi(s,a) = \{\phi(f) \mid f \in \mathcal{F}(s,a)\}$ denote the set of all features for which there exists at least one instantiation in $\mathcal{F}(s,a)$. Let $\mathcal{C}(f) = \{c_1, c_2, \ldots, c_k\}$ denote a set of $k \geq 0$ conditions, or propositions, that must hold in the state-action pair (s,a) for the feature instantiation f to be considered active. We may view every such condition $c_i = \langle \texttt{site}, \texttt{bit array}, \texttt{value}, \texttt{negated} \rangle$ as a four-tuple specifying:

1. The `site` for which we have a condition.

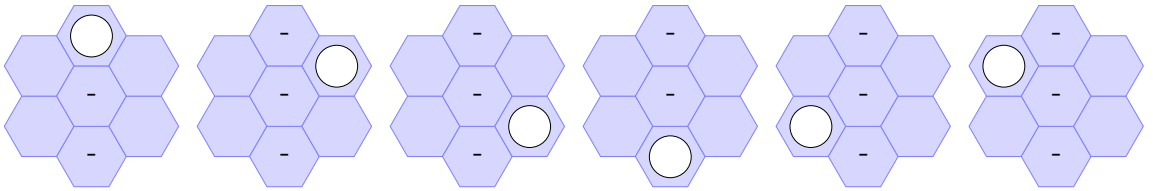


Fig. 16. Six different instantiations, each with the same position (centre of the depicted area) as anchor, of a single feature that requires the anchor to be empty (indicated by "**–**"), and an adjacent position to contain a white stone. This means that every instantiation is a conjunction of two conditions, where the condition for the empty position is identical for each of the instantiations. The feature is a disjunction of these six conjunctions.

2. The `bit array` (`empty`, `who`, or `what`) in which we look for a specific value.
3. The `value` that we check for in the specific site, mapping to a chunk, of the specific bit array.
4. Whether or not the condition should be `negated`; if true, the condition is satisfied if and only if the specified value is *not* found in the bit array.

A feature ϕ is considered active if and only if there exists at least one active instantiation: $\phi(s,a) = 1 \Leftrightarrow \exists_{f \in \mathcal{F}(s,a)} \big(\phi = \phi(f) \wedge \forall_{c \in \mathcal{C}(f)} [(s,a) \text{ satisfies } c]\big)$.

Since a feature instantiation is considered active if and only if all of its conditions are satisfied, it can be represented simply as a conjunction of its conditions: $C = c_1 \wedge c_2 \wedge \cdots \wedge c_k$. Since a feature only requires any one of its instantiations to be active for the feature to be considered active, the feature can be represented as a disjunction of several such conjunctions: $D = \big(c_i \wedge \ldots\ c_j\big) \vee \cdots \vee (c_k \wedge \cdots \wedge c_l)$. See Fig. 16 for an example.

We say that a conjunction (or feature instantiation) C_i *generalises* another conjunction C_j if and only if C_j contains all (and possibly more) conditions that C_i contains. In such a case, we have $\neg C_i \Rightarrow \neg C_j$, i.e. C_j will for sure not be satisfied if C_i is not satisfied. We say that a disjunction (or feature) D_i *generalises* another disjunction D_j if and only if D_j has at least one non-empty conjunction, and for every conjunction C_j in D_j, there exists a conjunction C_i in D_i such that C_i generalises C_j. In such a case, we have $\neg D_i \Rightarrow \neg D_j$, i.e. D_j will for sure not be satisfied if D_i is not satisfied. These relationships are one aspect that we will take into consideration when determining the order in which to evaluate propositions.

5.3. Implications between propositions based on domain knowledge

In addition to the domain-independent knowledge about generalisation relationships discussed above, we can leverage domain knowledge about implications between different propositions to further optimise the evaluation of features. In this subsection, we simply present the domain knowledge that is available. The way in which this is used is discussed afterwards. Note that in this case, "domain knowledge" refers to domain knowledge about the game state representations across all of Ludii [76]; this domain knowledge still generalises across many hundreds of games as they are modelled in Ludii.

The left column of Table 1 lists various propositions that feature instantiations may test for, and the right column lists other propositions that are automatically proven to be true whenever the matching propositions from the left column evaluate to true. For example, whenever a proposition that requires a site x to be empty evaluates to true, we can directly infer that that same position x is not owned by player 1 (or player 2 or any other player $p > 0$), that it is not occupied by a piece of type 1 (or type 2 or any other piece type $i > 0$), etc. Similarly, the negations of all the propositions in the right column can immediately be *disproven* whenever the matching propositions from the left column evaluate to true. Whenever a proposition a from the left column evaluates to false, it proves and disproves the propositions that correspond to the negation of a.

5.4. Organising instantiations and propositions in spatial pattern networks

Let $\mathcal{F}(s,a)$ denote a set of relevant feature instantiations for any arbitrary state-action pair (s,a), $\Phi(s,a) = \{\phi(f) \mid f \in \mathcal{F}(s,a)\}$ a set of features with at least one relevant instantiation, and $\mathcal{P}(s,a) = \bigcup\{\mathcal{C}(f) \mid f \in \mathcal{F}(s,a)\}$ a set containing all the propositions (or conditions) of all feature instantiations. The naive approach presented in Algorithm 1 would evaluate every proposition in $\mathcal{P}(s,a)$ at least once—and possibly many of them multiple times—to compute a feature vector $\boldsymbol{\phi}(s,a)$. We aim to construct a more efficient algorithm that:

1. Never evaluates the same proposition more than once.
2. Does not evaluate a feature instantiation f if its feature $\phi(f)$ is already known to be active (due to a different instantiation f' with $\phi(f') = \phi(f)$ already having been proven to be active).
3. Does not evaluate a proposition c for a feature instantiation f if there exists another instantiation f' such that f is generalised by f' (see Subsection 5.2), f' still needs to be evaluated (taking into account the previous point), and f' does not have c as a condition.
4. Leverages implications between propositions based on domain knowledge (see Subsection 5.3) to keep track of propositions that can be proven or disproven, without actually evaluating them.

Table 1
The left column lists propositions a that may be conditions of feature instantiations, where x always refers to a specific site. The right column lists propositions that can be proven by a; these can be automatically inferred to be true, without evaluating, whenever a has been evaluated to true.

Proposition a	Propositions proven by a
x is empty	x is empty
	x is not owned by player p (for any player $p > 0$)
	x is not piece i (for any $i > 0$)
x is not empty	x is not empty
x is owned by player p	x is owned by player p
	x is not piece i (for any i not owned by p)
	x is piece i (if i is the sole type owned by p)
	x is not empty
x is not owned by player p	x is not owned by player p
	x is not piece i (for any i owned by p)
x is piece i	x is piece i
	x is not empty
	x is not piece j (for any $j \neq i$)
	x is owned by player p (where p is the owner of i)
	x is not owned by player p (for any p that does not own i)
x is not piece i	x is not piece i
	x is not owned by player p (if i is the sole type owned by p)

Algorithm 2 Proposed algorithm for evaluation of active features.

Require: Feature instantiations $\mathcal{F}(s, a)$ represented as totally ordered list.
Require: Propositions $\mathcal{P}(s, a)$ represented as totally ordered list.
$\boldsymbol{\phi} \leftarrow \boldsymbol{\phi}_{init}$
`active_props` $\leftarrow \mathbf{1}^{|\mathcal{P}(s,a)|}$ // Bit array filled with 1 entries for every proposition.
`active_inst` $\leftarrow \mathbf{1}^{|\mathcal{F}(s,a)|}$ // Bit array filled with 1 entries for every instantiation.
for each set bit i in `active_inst` **do** // Evaluate i^{th} instantiation.
  for each proposition c of the i^{th} feature instantiation **do**
    if `active_props[`c`]` $= 0$ **then**
      continue // c has already been evaluated to true, so move on.
    end if
    `active_props[`c`]` $\leftarrow 0$
    if s violates c **then** // Condition not satisfied.
      DEDUCE(`active_props`, `active_inst`, $\neg c$)
      continue outer loop through `active_inst`
    else // s satisfies c
      DEDUCE(`active_props`, `active_inst`, c)
    end if
  end for
  $j \leftarrow$ feature index corresponding to the i^{th} feature instantiation
  $\boldsymbol{\phi}[j] \leftarrow 1$ // This feature has been proven to be active.
  for each other instantiation f of the same feature with index j **do**
    `active_inst[`f`]` $\leftarrow 0$ // Feature already active, so skip other instantiations.
  end for
end for
return feature vector $\boldsymbol{\phi}$

To this end, we propose the algorithm described in Algorithm 2, where it is assumed that a total ordering of all propositions in $\mathcal{P}(s, a)$, as well as total ordering of all feature instantiations in $\mathcal{F}(s, a)$, has already been computed. The way in which we compute these orderings is addressed in the next subsection. Furthermore, we assume that a vector $\boldsymbol{\phi}_{init}$ has already been precomputed, such that it contains entries of 1 for any features that can be guaranteed to be active irrespective of the game state s; these are features with at least one instantiation for which all conditions can already be satisfied at instantiation time. We assume that any feature instantiations for such features have also already been removed from $\mathcal{F}(s, a)$, since they are no longer necessary.

The basic premise of Algorithm 2 is that the `active_inst` bit array tracks feature instantiations that should still be (partially) evaluated, and `active_props` tracks which propositions have not yet been evaluated. At first, we assume that all instantiations should be evaluated, and do so in the order in which they have been sorted. Evaluating a feature instantiation is done by evaluating all of its propositions. As propositions and feature instantiations are evaluated, others may also already be deactivated (by assigning values of 0 in their respective bit arrays) and hence pruned. This is done by the DEDUCE() function described in Algorithm 3. Note that each of the loops that sets entries in bit arrays to 0 in this algorithm can be implemented to run at least partially in parallel by implementing them as bitwise ANDNOT operations.

Intuitively, this approach may be viewed as organising the feature instantiations and propositions into a network, with a variety of relationships between propositions and instantiations. See Fig. 17 for an example. Feature instantiations are

Algorithm 3 DEDUCE() function that makes deductions after evaluating a proposition.

Require: Bit array `active_props` of propositions that we may deactivate.
Require: Bit array `active_inst` of instantiations that we may deactivate.
Require: Proposition c that has been evaluated to true.
1: **for** each proposition c' such that $c \Rightarrow c'$ **do**
2: `active_props[`c'`]` $\leftarrow 0$ // c' already proven to be true.
3: **end for**
4: **for** each proposition c' such that $c \Rightarrow \neg c'$ **do**
5: **for** each instantiation f such that f requires c' **do**
6: `active_inst[`f`]` $\leftarrow 0$ // c', and hence f, already disproven.
7: **end for**
8: **end for**

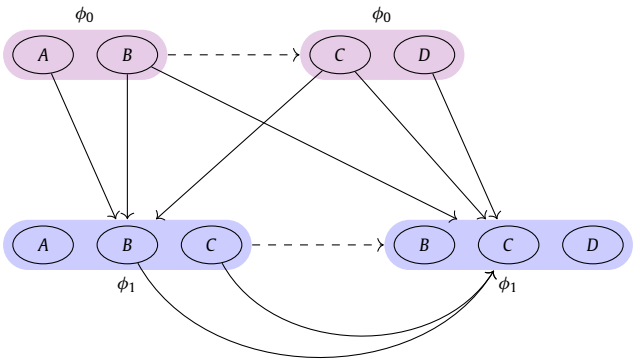


Fig. 17. A network representation of features, feature instantiations, and propositions. Circles represent propositions, labelled with letters A, B, C, D. Coloured boxes represent feature instantiations—conjunctions of the propositions they surround. Instantiations are labelled and coloured according to the feature they represent. In this example, there are two features—ϕ_0 and ϕ_1—with two instantiations each. A solid arrow from a proposition to an instantiation indicates that, if the proposition evaluates to false, the instantiation it points to is also disproven (in addition to the instantiation that the proposition is a part of itself). A dashed arrow from an instantiation f to another instantiation f' indicates that, if f is active, f' no longer needs to be evaluated.

evaluated by traversing the network in a fixed order; in the example figure, going from top-left to top-right to bottom-left to bottom-right. When a proposition evaluates to false, it can immediately deactivate any other complete feature instantiations that include the same proposition. When a feature instantiation evaluates to true, it can immediately deactivate any other complete feature instantiations that represent the same feature. Finally, when a proposition evaluates to true or false, it can prove or disprove other propositions or instantiations according to Table 1 (not included in the example figure). We refer to such a network as a *Spatial Pattern Network* (SPatterNet). We use a representation based on bit arrays and other primitive tables, rather than an explicit network representation, for improved performance. This is similar to how efficient implementations of Propositional Networks for GDL-based GGP [33,86] use table-based internal state representations [36].

5.5. Ordering propositions and instantiations

Given a set of relevant feature instantiations $\mathcal{F}(s,a)$, we aim to order feature instantiations and propositions in such a way that, in expectation, the number of evaluations of propositions required by Algorithm 2 for any arbitrary state-action pair (s,a) is minimised. The effectiveness of different pruning strategies (based on generalisation relationships between instantiations, different instantiations representing the same feature, and propositions proving or disproving other propositions) can be affected by these orderings in different ways. Hence, we start by considering only generalisation relationships between feature instantiations, and afterwards build on the resulting approach for ordering by incrementally taking into account the other strategies.

5.5.1. Ordering based on generalisation relationships

Let $f \in \mathcal{F}(s,a)$ and $f' \in \mathcal{F}(s,a)$ denote two different feature instantiations such that f' is generalised by f (i.e., f' is a conjunction of all propositions of f, plus at least one more). In such a case, f should always be fully evaluated before f'. This creates a trivial partial ordering of instantiations, where more general instantiations are always evaluated before more specific instantiations. An arbitrary ordering can be used between any pair of instantiations for which there exists no generalisation relationship.

5.5.2. Ordering instantiations based on shared features

Let $f \in \mathcal{F}(s,a)$ and $f' \in \mathcal{F}(s,a)$ denote two different feature instantiations such that both instantiations represent the same feature, i.e., $\phi(f) = \phi(f')$. When either one of these instantiations evaluates to true, evaluation of the other can be

skipped entirely, because the feature is already known to be active. Intuitively, this suggests that "shorter" instantiations (conjunctions of fewer propositions) should be ordered and evaluated before "longer" instantiations of the same feature. This intuition can easily be combined with the rule described above that more general instantiations should be evaluated before more specific instantiations. There are no conflicts between these two ideas, since a more general instantiation is also always shorter than a more specific instantiation. However, this intuition does not account for the observation that when a single proposition evaluates to false, this can immediately disprove any complete instantiation that it is a part of—including potentially many large or highly-generalised instantiations that would otherwise be deprioritised. This insight suggests that it may be useful to order individual propositions, rather than complete instantiations (conjunctions), and to prioritise propositions that appear in short conjunctions as well as propositions that appear in many conjunctions. These two criteria may conflict.

A similar conflict between two such heuristics commonly appears in boolean satisfiability (SAT) problems, in which the goal is to determine for any given propositional formula in conjunctive normal form, whether there exists an assignment of truth values to all variables such that the formula is true. Typical approaches for SAT problems use a backtracking search through the space of all possible assignments of truth values to variables [35,34]. In such a backtracking search, it is typically desirable to prioritise assigning truth values to the *most constrained* variables, because those are often the most likely to lead to unsatisfiable clauses—which allows for early backtracking. A popular set of heuristics is the set of *Maximum Occurrences in clauses of Minimum Size* (MOMS) heuristics [80], which prioritise variables that occur frequently, as well as variables that occur in small clauses (disjunctions). We take inspiration from these heuristics in our approach.

We propose an approach that orders propositions by iteratively picking propositions, sorting them in the order in which they are picked, keeping the following principles in mind:

1. If a disjunction (feature; disjunction of conjunctions of propositions) is not generalised by any other disjunctions, it must be evaluated.
2. If a disjunction must be evaluated, at least one of its conjunctions must be evaluated (repeated until either one conjunction evaluates to true, or all conjunctions evaluate to false).
3. If a conjunction must be evaluated, at least one of its propositions must be evaluated (repeated until either all its propositions evaluate to true, or one proposition evaluates to false).

Based on the first principle, we start out by splitting the set of all disjunctions (features) in two sets: a set $\mathcal{U}_0$ of *ungeneralised* disjunctions, and a set $\mathcal{G}_0$ of *generalised* disjunctions. For every ungeneralised disjunction $D \in \mathcal{U}_0$, there exists no other disjunction $D' \in \mathcal{U}_0 \cup \mathcal{G}_0$ such that D' generalises D. For every generalised disjunction $D \in \mathcal{G}_0$, there exists at least one ungeneralised disjunction $D' \in \mathcal{U}_0$ such that D' generalises D. For ease of reference, we list these and several other definitions of symbols in Table 2.

The second and third principles suggest that we should start out by picking a set of propositions such that at least one proposition from at least one conjunction from every ungeneralised disjunction $D \in \mathcal{U}_0$ is picked.[3] We partition the ungeneralised disjunctions $\mathcal{U}_0$ into subsets $\mathcal{U}_0^1, \mathcal{U}_0^2, \ldots$, such that $\mathcal{U}_0^i$ is the subset of all disjunctions that contain i conjunctions. As a special case, we start by immediately picking all propositions for conjunctions of length 1 in disjunctions of length 1 (i.e., disjunctions in $\mathcal{U}_0^1$). Afterwards, we loop through all $\mathcal{U}_0^i$ in increasing order of i, every time looping through the remaining uncovered disjunctions (in an arbitrary order) and picking a single proposition to cover that disjunction. Let $D \in \mathcal{U}_0^i$ denote such a disjunction that we need to pick a proposition from. Let $C(D)$ denote its set of conjunctions, $C \in C(D)$ one of the conjunctions, and $c \in C$ one of the propositions of such a conjunction. With some abuse of notation, we use $c \in C(D)$—with a lowercase c—to denote a proposition from any one of the conjunctions in $C(D)$. Let P denote the set of propositions that have already been picked. Let $|C|$ denote the length (i.e., number of propositions) of a conjunction C. Let Ω denote a set of all conjunctions C' such that C' is a part of some disjunction $D' \in \mathcal{U}_0^i$ where D' is not covered by any of the propositions picked so far in P. Let $\Omega_c = \{C \in \Omega \mid c \in C\}$ denote the subset of conjunctions in Ω that contain c as one of their propositions. Then, we pick the proposition $c^* \in C(D)$ given by Equation (1), which is based on the Jeroslow-Wang heuristic for SAT [47]:

$$c^* = \operatorname*{argmax}_{c \in C(D)} \sum_{C' \in \Omega_c} 2^{-|C'|} \tag{1}$$

Intuitively, this heuristic simply increases the score of a proposition for every conjunction that includes that proposition, while ignoring conjunctions from disjunctions that are already covered by any of the previously-picked propositions P. The exponent in the $2^{-|C'|}$ term favours propositions that appear in short conjunctions C' over propositions that appear in long conjunctions C'. Ties in the heuristic score are broken by computing a tie-breaker score in the same way over disjunctions in the set of generalised disjunctions $\mathcal{G}_0$ (instead of $\mathcal{U}_0$ in the definition of Ω), and any further ties are broken randomly.

[3] Picking a *minimal* set of such propositions would be equivalent to the hitting set problem, or set cover problem, which is NP-complete [48]. A common and simple heuristic in greedy algorithms to generate approximate solutions [26] would translate to our setting as a heuristic that would prioritise propositions that appear in a maximal number of disjunctions.

Table 2
Listing of symbols and definitions used throughout Subsubsection 5.5.2.

Symbol(s)	Definition
f, f'	Feature instantiations.
$\phi(f)$	Feature of which f is an instantiation.
D, D'	Disjunction (of conjunctions of propositions).
$\|D\|$	Cardinality (number of conjunctions) of a disjunction D.
$C(D)$	Set of conjunctions in the disjunction D.
C, C'	Conjunction of propositions.
$\|C\|$	Cardinality (number of propositions) of a conjunction C.
$c \in C$	One of the propositions of a conjunction C.
$c \in C(D)$	One of the propositions of any one of the conjunctions in $C(D)$.
$\mathcal{U}_0$	Set of ungeneralised disjunctions from which no conjunctions have been fully removed yet.
$\mathcal{G}_0$	Set of generalised disjunctions from which no conjunctions have been fully removed yet.
$\mathcal{U}_i$	Set of ungeneralised disjunctions from which i conjunctions have been fully removed already.
$\mathcal{G}_i$	Set of generalised disjunctions from which i conjunctions have been fully removed already.
$\mathcal{U}_j^i$	Subset of $\mathcal{U}_j$ containing only disjunctions that contain i conjunctions.
P	Set of propositions that have already been picked.
Ω	Set of all conjunctions C' such that C' is a part of some disjunction D' that, in turn, is part of a set $\mathcal{U}_i^j$ with the minimum i such that $\mathcal{U}_i \neq \varnothing$, such that D' is not covered by any propositions picked in P.
Ω_c	Subset of conjunctions C in Ω that contain c as one of their propositions.

After going through the procedure described above once, we end up with an initial list of propositions that cover at least one conjunction of every ungeneralised (and hence also every generalised) disjunction. Which propositions should be prioritised after this initial ordering can typically be very different depending on, for each of the propositions that have already been picked, whether they evaluate to true or false for any given state-action pair (s, a). This is because, whenever a proposition evaluates to false, any conjunction it is a part of becomes irrelevant. In contrast, whenever a proposition evaluates to true, any conjunction it is a part of becomes easier to prove, and hence arguably *more* important. This suggests that it may be desirable to construct a (binary) tree, rather than a single list, such that the order in which propositions are evaluated can be conditioned on the values that earlier propositions evaluate to. We explore this topic in more detail in Subsection 5.6, but in the remainder of this paper assume that we prefer only a single ordered list.

We make the simple assumption that all propositions picked so far would evaluate to true (even if that is typically not possible due to mutually exclusive propositions), and continue consecutive rounds of picking propositions based on that assumption. This is implemented by removing picked propositions from all conjunctions that contain them, deleting conjunctions that have a length of 0 after such removals from their disjunctions, and removing disjunctions that no longer have any conjunctions. After these changes, we re-compute which disjunctions are generalised or ungeneralised. At this stage, disjunctions are not split up in only two sets $\mathcal{U}_0$ and $\mathcal{G}_0$, but potentially many sets $\mathcal{U}_i$ and $\mathcal{G}_i$ for $i \geq 0$. A set with subscript i contains any disjunctions from which i full conjunctions have already been removed due to being fully covered. Where we previously described that we would loop through $\mathcal{U}_0$ to pick new propositions, we now loop through $\mathcal{U}_i$ for the minimal i such that $\mathcal{U}_i \neq \varnothing$. When applying the heuristic score of Equation (1), all $\mathcal{U}_i$ in increasing order of i, followed by all $\mathcal{G}_i$, are used as tie-breakers in the definition of Ω. Intuitively, we prioritise disjunctions with fewer fully covered conjunctions over disjunctions with more fully covered conjunctions, and place more importance on ungeneralised disjunctions than generalised disjunctions. This complete process is repeated until all propositions have been picked. Algorithm 4 provides pseudocode of the entire algorithm to order propositions.

After obtaining a total ordering of the propositions from this process, we order the feature instantiations such that no instantiation f is ordered before another instantiation f' if f requires a proposition that is ordered after all propositions required by f'. In other words, the last-ordered proposition of an instantiation determines the ordering of that instantiation. This lets us use Algorithm 2—which is based on looping through an ordered list of instantiations—while still ensuring that the propositions are evaluated in approximately the order that they were sorted in.

5.5.3. Ordering based on implications between propositions

As a final step, we consider taking into account the implication relationships between various propositions, as described in Subsection 5.3, when ordering feature instantiations. Whenever a proposition c is evaluated, it will evaluate to either `true` or `false`, and in either case there can be several other propositions c' that will be proven ($c \Rightarrow c'$ or $\neg c \Rightarrow c'$) or disproven ($c \Rightarrow \neg c'$ or $\neg c \Rightarrow \neg c'$). We avoid making any additional assumptions concerning the marginal likelihoods for any given proposition to be true or false (which would require game-specific domain knowledge), instead simply assuming that any proposition is equally likely to evaluate to either true or false. This means that, when a procedure like the one described above picks a proposition c to be inserted in the ordered list, we expect there to be a probability of 0.5 to automatically gain information on any propositions that are proven or disproven by c being true, and a probability of 0.5 to automatically gain information on any propositions that are proven or disproven by c being false.

We incorporate these ideas in the ordering procedure by adapting the heuristic of Equation (1) by adding 50% of the heuristic score of any proposition c' that may be proven or disproven by the evaluation of another proposition c, to the heuristic score of c. More formally, let $\top(c)$ denote the set of propositions that are either proven or disproven when c

Algorithm 4 Order propositions of feature set (see Table 2 for definitions of symbols).

Require: Initial set of ungeneralised disjunctions $\mathcal{U}_0$
Require: Initial set of generalised disjunctions $\mathcal{G}_0$

function OrderPropositions
  $\mathcal{U}_i \leftarrow \varnothing$ for all $i > 0$
  $\mathcal{G}_i \leftarrow \varnothing$ for all $i > 0$
  `ordered_propositions` $\leftarrow$ `[]` // Init as empty list. P is set representation of list.
  while at least one $\mathcal{U}_i \neq \varnothing$ **do**
    $i \leftarrow \min_i$ such that $\mathcal{U}_i \neq \varnothing$
    Partition $\mathcal{U}_i$ into subsets $\mathcal{U}_i^j$ of disjunctions with $j \geq 1$ conjunctions
    for $j = 1, 2, 3, \ldots$ **do**
      for each disjunction $D \in \mathcal{U}_i^j$ **do**
        if D not already covered by P **then**
          $c \leftarrow$ PickProposition(D, i, j)
          Append c to `ordered_propositions`
        end if
      end for
    end for
    for all i' **do**
      for all disjunctions D in $\mathcal{U}_{i'}$ or $\mathcal{G}_{i'}$ **do**
        for all conjunctions $C \in C(D)$ **do**
          Remove all picked propositions P from C
          if $|C| = 0$ **then**
            Remove C from $C(D)$
          end if
        end for
        if $|D| = 0$ **then**
          Remove D from $\mathcal{U}_{i'}$ or $\mathcal{G}_{i'}$
        end if
      end for
    end for
    Re-compute all $\mathcal{U}_{i'}$ and $\mathcal{G}_{i'}$ // See Table 2.
  end while
  return `ordered_propositions`
end function

function PickProposition(D, i, j)
  Compute set of conjunctions Ω from $\mathcal{U}_i^j$. // See Table 2.
  Pick $c \in C(D)$ that maximises heuristic. // Equation (1) or Equation (2)
  Break ties using $\mathcal{U}_i^{j+1}, \mathcal{U}_i^{j+2}, \ldots, \mathcal{U}_{i+1}^1, \mathcal{U}_{i+1}^2, \ldots, \mathcal{G}_0^1, \mathcal{G}_0^2, \ldots, \mathcal{G}_1^1, \ldots$
  return c
end function

evaluates to true, and let $\bot(c)$ denote the set of propositions that are either proven or disproven when c evaluates to false. Then we replace the original heuristic of Equation (1) by the updated version in Equation (2):

$$c^* = \operatorname*{argmax}_{c \in C(D)} \sum_{C' \in \Omega_c} 2^{-|C'|} + \frac{1}{2} \sum_{c' \in \top(c)} \left[\sum_{C' \in \Omega_{c'}} 2^{-|C'|} \right] + \frac{1}{2} \sum_{c' \in \bot(c)} \left[\sum_{C' \in \Omega_{c'}} 2^{-|C'|} \right] \tag{2}$$

5.6. Discussion

In Subsubsection 5.5.2, we remarked that in theory, it may be preferably to order propositions in a binary tree rather than a list, such that the order in which later propositions are evaluated can be conditional on the truth values that earlier propositions evaluate to. If an earlier proposition evaluates to false, any conjunction it is a part of is immediately disproven and therefore becomes irrelevant, which should in turn deprioritise any other propositions in the same conjunctions. Conversely, if an earlier proposition evaluates to true, any conjunctions it is a part of are closer to being proven, which should raise the priority level of other propositions in those conjunctions.

Intuitively, this also happens when MOMS heuristics are used to select variables to focus on next in a backtracking search for SAT problems. During a backtracking search, truth values are assigned to selected variables, and the heuristics are used "online" (during the search) to select a single next variable for that specific part of the search tree—which corresponds to a specific assignment of truth values to previously-selected variables.

The core reason not to consider such an approach further in this paper is that it would have excessive memory requirements. In contrast to SAT problems where the selection of variables happens online during a backtracking search, we deal with an offline preprocessing step of which the result (ordering of propositions) remains stored in memory for future use (to speed up online feature evaluations during gameplay). Let $K \geq 0$ denote the number of propositions for a full feature

set. A single ordering of these propositions in a list only contains K entries, but an ordering in a binary tree—allowing for later propositions to be conditioned on truth values of earlier propositions—would have $2^K - 1$ entries.

6. Experiments

In this section, we describe several experiments used to evaluate the performance of the proposed approach for evaluating active features using SPatterNets. In our experiments, we consider the following four approaches for evaluating active features:

1. **Naive**: a straightforward, naive approach for evaluating active features, as described by Algorithm 1. This represents the simplest baseline.
2. **Tree**: an approach that organises feature instantiations in a tree, such that more specific instantiations are located below more general instantiations. When an instantiation evaluates to false, any (more specific) instantiations below it are skipped. This is similar to the approaches used, for example, by Levinson & Snyder [55] and Müller [68,69], although likely less efficient due to our requirements for a general system that is not specific to a single game or board geometry. It also bears resemblance to the DAG-based approach described by Buro [20]. More precisely, this approach incrementally builds up a tree by inserting feature instantiations one by one, always inserting any given new instance as a child of whichever potential parent (generaliser) already has the deepest current position in the tree. This represents a more advanced baseline.
3. **SPatterNet**: the main approach proposed in this paper, which organises propositions and instantiations as described in Subsection 5.5, and evaluates active features using Algorithm 2.
4. **SPatterNet (JIT)**: A variant of the SPatterNet approach, in which features are instantiated in a just-in-time (JIT) manner when needed, rather than instantiating all features for all possible move keys (as described in Subsection 5.1) in advance. This can significantly reduce the initialisation time required before a game can start being played. Not generating instantiations for moves that are never legal in practical play can also reduce memory usage and the sizes of hash maps, which in turn may be a benefit in terms of processing speed. Because in some cases it will be necessary to instantiate features at runtime (while a game is being played), there may also be a reduction in processing speed.

All experiments are run using a set of 33 highly varied, distinct games, which we provide additional details on in Appendix A. All experiments are run using the Ludii general game system. The source code of Ludii, as well as all the code for spatial features, is available from https://github.com/Ludeme/Ludii. Version 1.3.1 of Ludii was used for the experiments described in this paper.

The primary measure of performance we focus on is the number of playouts per second we can run. Playouts are run from the initial game state until terminal game states are reached, by sampling actions uniformly at random (unless stated otherwise), while computing the active features for every legal action in every state. For every game, every feature set, and each of the evaluated implementations, we run a separate process with access to two cores of a 2.6 GHz Intel Xeon E5-2690 v3 CPU, and 4096 MB allocated to the Java Virtual Machine (JVM). Playouts are run sequentially, but access to a second core may be used, for instance, by Java's garbage collector. Every process uses 60 seconds of warming up time for the JVM (during which the JIT variant of SPatterNet can also instantiate features), after which the number of playouts per second is measured over a duration of 600 seconds.

6.1. Atomic feature sets

For cases where it is infeasible or undesirable to exhaustively generate all patterns or features up to a given size (as commonly done in Go programs), several approaches have been proposed that start with a smaller set of *atomic* features, and gradually grow this by generating more complex features that are composed of two or more other (atomic, or previously-generated composite) features [20,101,92,95]. For our first experiment, we evaluate the performance on such atomic feature sets.

Using the spatial feature format as described in Subsection 4.3, we define atomic features to be features that have exactly one requirement for the state (i.e., one walk with one element such as "must be empty" or "must not be enemy"), in addition to any specifiers related to actions. We consider several different atomic feature sets, described as Atomic-M-N for different integer values $M \geq 1$ and $N \geq M$. In a set labelled Atomic-M-N, we generate all such features for any walk restricted to a length of up to M steps, allowing walks of up to $N \geq M$ steps for "straight" walks (which only have steps with rotation values $\rho = 0$, i.e. no turns). Furthermore, the following rules apply for generating atomic feature sets:

- Let K denote the largest number of orthogonal connections (including artificial off-board connections) for any site in a given game. In the vast majority of games, this number is the same for *all* sites (due to the prevalence of boards based on regular tilings). In all generated walks, we only consider rotation values that are fractions of K (i.e. $\rho = \frac{1}{K}$, $\rho = \frac{2}{K}$, $\ldots$, $\rho = 1$).
- Every feature must have at least a `from` or a `to` position specified (or both). In games for which the rules can never generate any moves with a distinct `from` position, no features with `from` or `last_from` specifiers are generated.

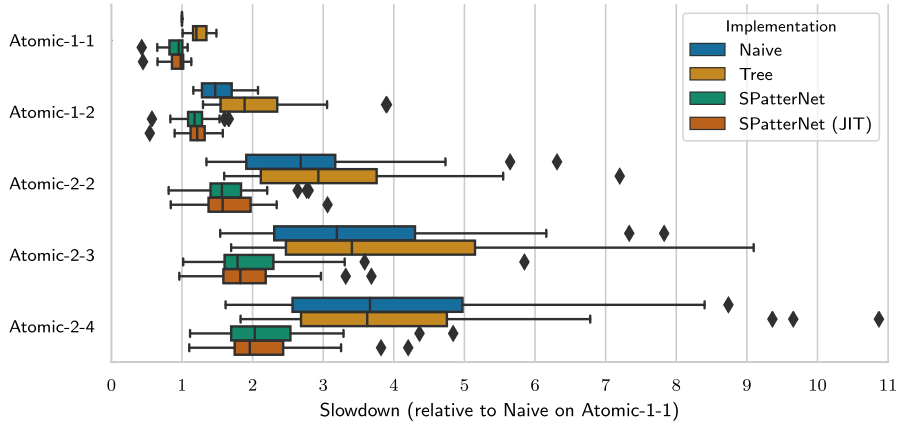


Fig. 18. Boxplots summarising slowdowns of several feature evaluation implementations on several different sets of atomic features. Every boxplot summarises 33 data points, from 33 different games. Lower slowdown values are better (with values < 1 being speedups). The size (number of features) per feature set increases as we go down along the y-axis.

- For any generated feature, either the `from` or the `to` position (or both) must have a walk of length 0, causing it to overlap with the anchor.
- In 2-player games with only one piece type per player (Hex, Go, Tic-Tac-Toe, etc.), no features are generated with elements testing for piece types—only testing for friend or enemy is sufficient in these games.

6.1.1. Results

The raw number of playouts per second varies significantly between different games, which means that the effect of evaluating features on those playout rates cannot be directly compared or averaged across games. To present aggregate results, we take the number of playouts per second for the smallest feature set—Atomic-1-1—with the simplest feature evaluation approach—Naive—as a baseline b per game. For every other pair of feature set and feature evaluation approach, for every game, we compute the *slowdown* by dividing the per-game baseline b by that pair's raw number of playouts per second. Larger values indicate a greater slowdown (relative to Naive with the smallest Atomic-1-1 feature set) as a result of using a specific combination of a feature set and feature evaluation approach. Values below 1.0 are speedups.

The boxplots in Fig. 18 summarise these results, with every boxplot—one for every possible pair of an atomic feature set and a feature evaluation approach—representing 33 data points for 33 different games. Every atomic feature set is a strict subset of all the ones below it, which means that feature sets that appear lower in the figure will always require at least as much work to evaluate active features as those that are higher in the figure—and therefore have greater slowdowns. In every boxplot, the vertical bar represents the median result (over 33 games), the coloured box covers the interquartile range, and the whiskers cover any remaining data up to a distance of 1.5 times the interquartile range below or above the 25^{th} or 75^{th} percentile, respectively. Data points outside of the whiskers are visualised individually as diamonds.

For each of the 33 games, and for each of the atomic feature sets, we rank the four feature evaluation approaches based on their performance for that specific game and atomic feature set. This means that, for each atomic feature set, we distribute ranks ranging from 1 (best) to 4 (worst) among the four evaluated approaches 33 times. For each approach, on each feature set, the frequency per rank (number of times—out of 33—that a certain rank was assigned) is listed in Table 3.

6.1.2. Discussion

The boxplots of Fig. 18 consistently show, across all considered feature sets, that both variants of SPatterNet are the two most efficient implementations in terms of aggregate results such as the median slowdowns (or speedups), the most extreme outliers, etc. The advantage of the SPatterNet approaches over the other two approaches arguably becomes more pronounced as the size of the feature sets increases.

Interestingly, the Tree approach appears to be outperformed by the Naive approach, in particular on the smallest feature sets, even though the Tree approach is meant to be an optimisation based on generalisation relationships between feature instantiations. We remark that in atomic feature sets, there are generally relatively few features or feature instantiations that actually generalise others; by design the atomic features are meant to be mostly independent features with little overlap. This is particularly true for the smallest sets; in the larger atomic feature sets, which allow for longer walks, it is more likely that multiple different walks (including some turns) will have overlapping destinations. Therefore, in the smaller atomic feature sets, the Tree approach is slower than the Naive approach, because its generalisation-based optimisation is ineffective, but it still has a more complex implementation with additional overhead. In the larger atomic feature sets, it appears that this optimisation becomes closer to worthwhile. The SPatterNet approaches include similar generalisation-based optimisations (which are ineffective in small atomic feature sets), but also other optimisations that can already pay off for smaller feature sets.

Table 3
For each of the 33 games, and for every feature set, we rank the four feature evaluation approaches based on their performance. This table lists, for every feature set, how often each rank was obtained by every feature evaluation approach, where a rank of 1 means best performance and a rank of 4 means worst performance in a given game.

	Frequency per Rank			
	Rank 1	Rank 2	Rank 3	Rank 4
Atomic-1-1				
Naive	8	6	19	0
Tree	0	0	0	33
SPatterNet	21	6	6	0
SPatterNet (JIT)	4	21	8	0
Atomic-1-2				
Naive	1	3	28	1
Tree	0	0	1	32
SPatterNet	19	12	2	0
SPatterNet (JIT)	13	18	2	0
Atomic-2-2				
Naive	0	3	23	7
Tree	0	0	7	26
SPatterNet	19	12	2	0
SPatterNet (JIT)	14	18	1	0
Atomic-2-3				
Naive	0	0	27	6
Tree	0	0	6	27
SPatterNet	18	15	0	0
SPatterNet (JIT)	15	18	0	0
Atomic-2-4				
Naive	0	0	24	9
Tree	0	0	9	24
SPatterNet	13	20	0	0
SPatterNet (JIT)	20	13	0	0

The per-game comparisons summarised by Table 3 lead to similar conclusions. The two SPatterNet approaches already share the top 2 ranks in the majority of games even for the smallest atomic feature set, and in the largest feature sets they entirely dominate the top 2 ranks. Furthermore, neither of them has the worst performance (rank 4) in any case. Between the two SPatterNet approaches, the results in the table suggest that the standard variant may perform better for smaller feature sets, whereas the JIT variant may perform better for larger feature sets. However, considering the results depicted by Fig. 18, such differences are unlikely to be large in magnitude.

6.2. Trained feature sets

In our next experiment, we move on to benchmarking larger, trained feature sets which include non-atomic composite features that have been discovered to be potentially useful in a training process. In each game, we start out with its Atomic-2-4 feature set—the largest of the atomic feature sets evaluated in the previous experiment. We interleave [108,95] policy training and feature discovery (the addition of new features) in a self-play training setup between MCTS agents that are guided by the trained policy [2,90], where the policy uses the features as input. For every game, we run such a training process for up to 200 episodes, or up to 24 hours. After every self-play episode, we attempt to add one proactive and one reactive feature to the feature set, by combining two existing feature instantiations into a new composite feature [95]. This means that, in comparison to the Atomic-2-4 sets, each of our feature sets grows by up to 400 features (of which 200 proactive and 200 reactive). More detailed information on this training setup is provided in Appendix B.

After such a training run for a game, we do not only have a new, larger feature set, but also a trained policy π which can, for any given state-action pair (s, a), provide a probability $0 \leq \pi(s, a) \leq 1$ with which the action a should be selected in state s. For each of the large, trained feature sets, we benchmark the number of playouts per second for every game two times; once by still sampling actions uniformly (evaluating active features solely for the sake of benchmarking performance), and once by sampling actions according to the trained policy π.

6.2.1. Results

Fig. 19 depicts boxplots that summarise the slowdowns resulting from the trained feature sets. Note that we still use the same baseline—the playouts per second from the Naive approach on the smallest atomic feature set—as in Fig. 18 to compute the relative slowdowns. Table 4 lists for each approach how often, out of 33 games, it achieved each of the ranks ranging from 1 (best) to 4 (worst).

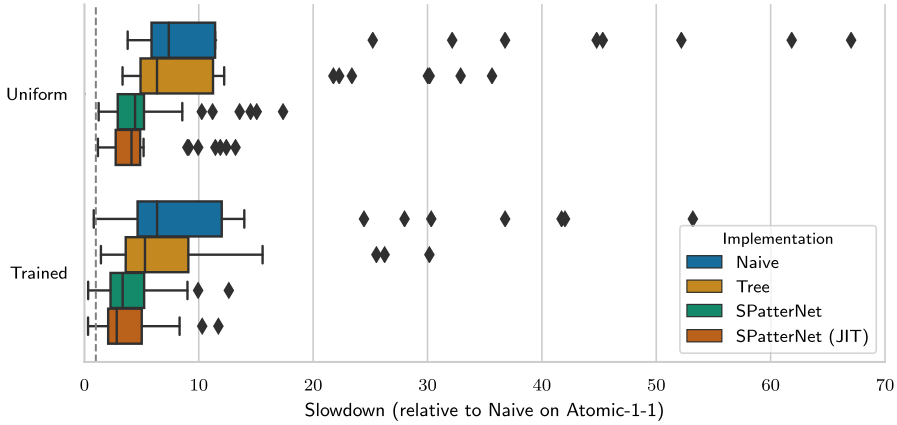


Fig. 19. Boxplots summarising slowdowns of several feature evaluation implementations on large, trained feature sets; once where actions are sampled uniformly at random, and once where actions are sampled according to a trained policy. Every boxplot summarises 33 data points, from 33 different games. Lower slowdown values are better (with values < 1 being speedups—the dashed vertical line indicates this point).

Table 4
For each of the 33 games, and for both uniform action sampling as well as action sampling from a trained policy, we rank the four feature evaluation approaches based on their performance. This table lists how often each rank was obtained by every feature evaluation approach, where a rank of 1 means best performance and a rank of 4 means worst performance in a given game.

	Frequency per Rank			
	Rank 1	Rank 2	Rank 3	Rank 4
Uniform Action Sampling				
Naive	0	0	5	28
Tree	0	0	28	5
SPatterNet	3	30	0	0
SPatterNet (JIT)	30	3	0	0
Trained Policy Action Sampling				
Naive	0	0	6	27
Tree	0	0	27	6
SPatterNet	9	24	0	0
SPatterNet (JIT)	24	9	0	0

6.2.2. Discussion

Both Fig. 19 and Table 4 show continuations of the trends discussed in Subsubsection 6.1.2. The two variants of SPatterNets outperform the others in terms of aggregate results, as well as dominating the top 2 ranks across all 33 games. There also appears to be a more pronounced advantage for the JIT variant over the standard variant of SPatterNet in the trained feature sets. The Tree approach appears to more convincingly outperform the Naive approach (especially in Table 4) than it does in the atomic feature sets. Its generalisation-based optimisation is significantly more effective in the trained feature sets, because every new feature that is added during the training process is a composite of—and therefore generalised by—at least two other feature instantiations.

We do not observe any major differences in these trends between the results gathered from uniformly random playouts, and playouts where actions are sampled from trained policies. On average playouts appear to run more quickly when actions are sampled from the trained policy, but there are also games where they are slower. These differences are observed regardless of which implementation is used for evaluating active features, and are due to overall more "intelligent" action selection—which in many games causes playouts to have a shorter duration in terms of the average number of actions per playout.

6.3. Playing strength of MCTS with trained policies

The purpose of the final experiment for this paper is to evaluate the impact that the improved speed of SPatterNet over the baselines has in terms of the playing strength of MCTS agents that are guided by trained policies. Taking the same trained feature sets as used in Subsection 6.2, a separate MCTS agent is constructed for each of the four feature evaluation approaches under consideration (Naive, Tree, SPatterNet, and SPatterNet (JIT)). Except for using different approaches to evaluate features, these four agents are identical. The rest of the setup of the biased MCTS agents is as described in Appendix B for the agents used during self-play training, with the only exception being that, in evaluation games—as opposed to training games—the agents pick actions that maximise visit counts, rather than proportionally to visit counts. For each

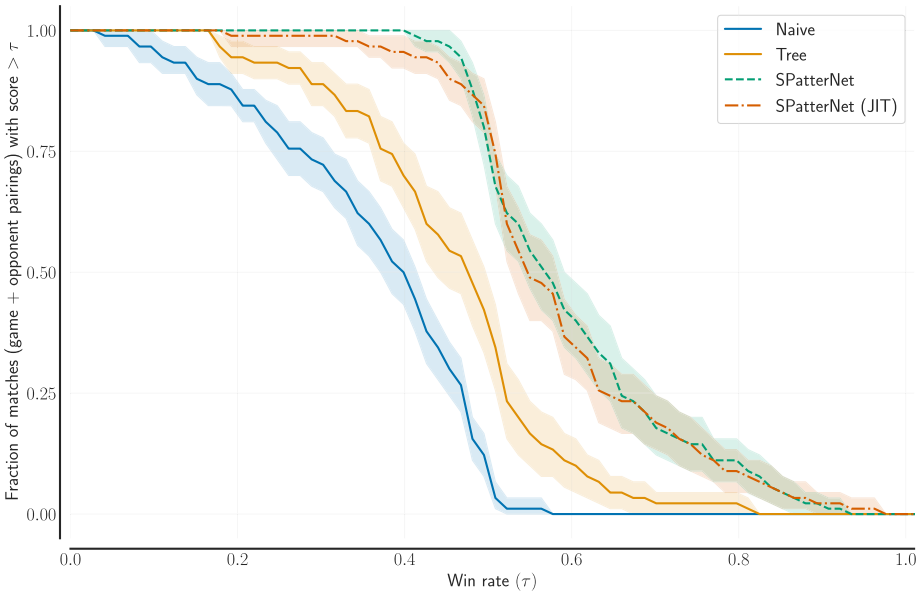


Fig. 20. Performance profiles [1] for MCTS agents biased by trained features, with different approaches for evaluating features.

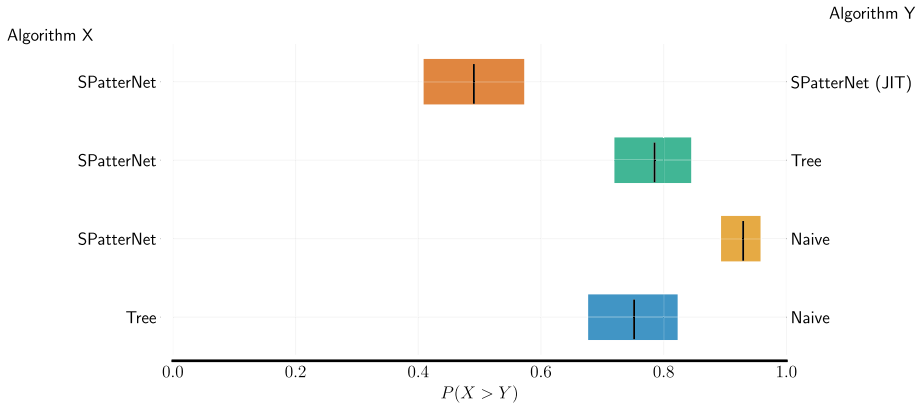


Fig. 21. Pairwise probabilities of improvement [1] for algorithms on the left-hand side to outperform algorithms on the right-hand side.

of the six possible pairings of four such agents (excluding mirror matchups, excluding pairings that are permutations of other pairings), 100 evaluation games were played between the two agents, in each of the 30 two-player games also used in previous experiments (excluding three games for more than two players). The three games for more than two players are excluded from this experiment, because they require a significantly larger amount of computation resource for appropriate statistical analyses. This is due to the large number of possible permutations of agent assignments to player roles in games with more than two roles.

6.3.1. Results

Fig. 20 depicts performance profiles [1] for the MCTS agents using different approaches for their feature evaluations. A datapoint at (x, y) in the plot means that an agent achieved a winrate (averaged over the four possible opponents, including mirror matchups) greater than or equal to x on a fraction equal to y out of the thirty games. The shaded intervals depict 95% bootstrap confidence intervals from 2000 replicates, indicating variability in results over the four different opponents for every agent.

Fig. 21 depicts probabilities of improvement for four different pairs of agents. For each of the pairs (rows), it provides the probability that the approach on the left-hand side improves upon the approach on the right-hand side on any given individual game from the set of thirty games, using the implementation from the *rliable* framework [1], which uses the Mann-Whitney U-statistic [62]. Confidence intervals are based on 2000 bootstrap replications.

Table 5 lists 95% confidence intervals for the median, interquartile mean (IQM), and mean win rates of the four evaluated algorithms, each against the three other algorithms over the 30 different games. Note that the point estimates for the means

Table 5
95% bootstrap confidence intervals for median, interquartile mean (IQM), and mean win rates (stratified across 30 games) of MCTS agents biased by trained features, with different approaches for evaluating features.

Algorithm	Win rate against all other algorithms		
	Median	IQM	Mean
Naive	[0.35, 0.41]	[0.37, 0.41]	[0.35, 0.38]
Tree	[0.45, 0.49]	[0.44, 0.48]	[0.43, 0.48]
SPatterNet	[0.56, 0.61]	[0.55, 0.60]	[0.58, 0.62]
SPatterNet (JIT)	[0.54, 0.59]	[0.54, 0.58]	[0.57, 0.60]

and medians correspond to the areas under the curves and the points of intersection with the horizontal $y = 0.5$ line, respectively, of the performance profiles in Fig. 20.

6.3.2. Discussion

The performance profiles of Fig. 20 show both variants of SPatterNet (the regular and the JIT variant) *stochastically dominating* [56,37] both of the baselines (with their plots not being below either of the baselines for any point τ), which is a clear sign of them providing consistently higher levels of performance. Fig. 21 similarly shows the SPatterNet approach outperforming both of the baselines by statistically significant margins (with the Tree baseline itself also significantly outperforming the Naive baseline). While the standard variant of the SpatterNet approach appears to possibly slightly outperform the JIT variant in the performance profiles, this is not by a significant margin, and also cannot be concluded from the pairwise comparison in Fig. 21. In the latter figure, with the upper thresholds of the confidence intervals exceeding 0.75 for all three of the bottom rows, these results are also statistically meaningful following the threshold recommended by Bouthillier et al. [6]. The results in Table 5 lead to similar conclusions. For both SPatterNet and SPatterNet (JIT), even the lower bounds of all three aggregate metrics exceed the corresponding upper bounds for the two baseline algorithms, demonstrating statistically significant levels of improvement.

7. Applications beyond games

The focus in this paper was on efficiently evaluating patterns of spatial state-action features for game playing as an application domain. However, the SPatterNet approach developed for this purpose applies to a highly general and abstract formulation of the problem, where we simply aim to optimise the order in which propositions of logical formulas in disjunctive normal form are evaluated. This problem formulation, and hence also the SPatterNet approach, may be applicable to other types of features than spatial ones, or other types of problems than game playing.

For example, potential other applications may include classification domains with costly features, such as medical, fraud detection, or computer security domains, where feature values can be costly to obtain [63,106,46]. In such domains, sets of learned decision rules may be thought of as logical formulas in disjunctive normal form, where every conjunction corresponds to a single rule, and a disjunction is formed by a set of different rules that have the same output class [63].

Aside from sets of decision rules, models such as Decision Trees or Random Forests also lend themselves naturally towards being translated into disjunctive normal form. Such models have also been used in settings with costly features [72,73]. In a Decision Tree, every path from the root to a leaf may be viewed as a conjunction, and sets of paths leading to different leaves with the same output class can be viewed as disjunctions of conjunctions. While the optimal order in which to evaluate propositions would likely closely correspond to the order in which they appear in a Decision Tree, it may deviate if there are implication relationships between propositions, and propositions that appear in multiple subtrees below a shared ancestor. For Random Forests with, for example, a majority voting rule, an efficient ordering for propositions may also differ from the order dictated by any single individual tree of the forest.

8. Conclusion

In previous work, we proposed an initial design and formalisation of spatial features for general games [15] in the Ludii general game system, and demonstrated that they can be used to train simple policies that effectively improve playing strength in a variety of different games from self-play using few resources [95,96]. However, whether or not such policies substantially improve playing strength also depends greatly on the amount of computational overhead associated with feature evaluations [95]. This paper has two core contributions. Firstly, we extend the design and formalisation of the spatial features from previous work, and provide significantly more detail on several implementation and design decisions. Combined with the publicly available source code, this should aid replicability. Secondly, we propose a new approach to significantly improve the efficiency of evaluating active features, reducing the computational overhead when using features to, for instance, guide a tree search. An empirical evaluation demonstrates the improvements in efficiency, and also shows that this significantly improves the playing strength of agents using the features to guide their search.

Within the domain of AI for games, one idea for future research is transfer learning of policies based on spatial state-action features between games. The features have been designed to facilitate this—for instance by formalising the walks

used to define relative positions in such a way that their semantics remain similar when transferred to different board geometries—but this has yet to be evaluated. Another idea for future research would be a further extension of the feature formalisation, to include additional types of elements that may be important for categories of games that are not yet well-supported. For example, games with hidden information would likely benefit from an extended format where features can specify that certain positions should be hidden, and games where pieces can stack up on top of each other would benefit from adjustments to take such a third "spatial dimension" into account. These are some of the same categories of games that Soemers et al. [93] also identified as having received little attention in research based on deep learning approaches.

While we focused on applications for game AI in this paper, we remark that the proposed SPatterNet approach for organising propositions and determining the order in which they should be evaluated is not necessarily restricted to game AI. A similar approach may more generally be applicable to any domain where sets of disjunctions of conjunctions need to be evaluated efficiently, in particular when there are generalisation relationships or implications (based on domain knowledge) between them that the SPatterNet approach can leverage.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

A URL for the GitHub repository containing the code is provided in the paper.

Acknowledgements

This research is funded by the European Research Council as part of the Digital Ludeme Project (ERC Consolidator Grant #771292) led by Cameron Browne at Maastricht University's Department of Advanced Computing Sciences. We wish to thank Walter Crist, Cédric Piette, Chiara Sironi, and Mark Winands for helpful pointers to related work. This work was carried out on the Dutch national e-infrastructure with the support of SURF Cooperative (grant no. EINF-1133). This work used the Dutch national e-infrastructure with the support of the SURF cooperative using grant no. EINF-4028. This publication is part of the project "Evaluation of Trained AIs for General Game Playing" (with project number EINF-4028) of the research programme Computing Time on National Computer Facilities which is (partly) financed by the Dutch Research Council (NWO).

Appendix A. Games used for experiments

This appendix provides additional detail on all the games (a total of 33 distinct games) used for experiments described throughout this paper. Table A.6 lists the names of all the selected games, as well as several properties. Fig. A.22 depicts thumbnails of all the games, which provides an impression of the variety in board shapes and connectivity structures included in the experiments. This selection of games includes:

- Several games that have been commonly used as benchmarks for AI research, such as Arimaa [104], Chess [21], Go [70,88], and Hex [25].
- A mix of games where actions primarily involve movement from one site to another, and games where actions primarily involve placing new pieces on sites (see Table A.6)—these are significant differences for the spatial features discussed in this paper.
- Several games with a significant degree of asymmetry (in initial setup of pieces, piece types, victory conditions, etc.) between different players; ArdRi, Bizingo, Fox and Geese, and Tablut.
- Two stochastic games: Royal Game of Ur, and XII Scripta.
- Three games with more than two players: Chinese Checkers, Level Chess, and Triad.
- A high degree of variety in board shapes and connectivity structures between sites; see Fig. A.22.

For all games, we used the default options (e.g., default board sizes) as implemented in the Ludii general game system. Detailed information on each of these games can be found on https://ludii.games/library.php.

Appendix B. Details on self-play training setup

Like the speed evaluations described in Section 6, every training run (one per game) ran as a separate process on two cores of a 2.6 GHz Intel Xeon E5-2690 v3 CPU, with 4096 MB of RAM allocated to the Java Virtual Machine (JVM), using Java version 8u261. Every training process ran for up to 200 episodes, or up to 24 hours of wall time. The training process is similar as described in our previous work [95,96], which in turn is largely inspired by the AlphaGo Zero and Expert Iteration training processes [2,90]. Additional details are provided below.

Table A.6
Various properties of the games used in experiments. In the second column, "Movement" is listed for games that involve actions with both source and destination positions (steps, slides, hops, etc.), and "Placement" is listed for games that involve actions with only a destination position (e.g., placing stones). The final column (K) lists the number of players per game.

Game	Primary Action Types	Asymmetric	Stochastic	K
Alquerque	Movement	×	×	2
Amazons	Movement & Placement	×	×	2
ArdRi	Movement	✓	×	2
Arimaa	Movement	×	×	2
Ataxx	Movement	×	×	2
Bao Ki Arabu (Zanzibar 1)	Placement	×	×	2
Bizingo	Movement	✓	×	2
Breakthrough	Movement	×	×	2
Chess	Movement	×	×	2
Chinese Checkers	Movement	×	×	6
English Draughts	Movement	×	×	2
Fanorona	Movement	×	×	2
Fox and Geese	Movement	✓	×	2
Go	Placement	×	×	2
Gomoku	Placement	×	×	2
Gonnect	Placement	×	×	2
Havannah	Placement	×	×	2
Hex	Placement	×	×	2
Kensington	Movement	×	×	2
Knightthrough	Placement	×	×	2
Konane	Movement	×	×	2
Level Chess	Movement	×	×	4
Lines of Action	Movement	×	×	2
Pentalath	Placement	×	×	2
Pretwa	Movement	×	×	2
Reversi	Placement	×	×	2
Royal Game of Ur	Movement	×	✓	2
Shobu	Movement	×	×	2
Surakarta	Movement	×	×	2
Tablut	Movement	✓	×	2
Triad	Movement & Placement	×	×	3
XII Scripta	Movement	×	✓	2
Yavalath	Placement	×	×	2

Each training process starts by generating the Atomic-2-4 feature sets for each of K players in a given K-player game. Self-play experience is generated by K copies of MCTS agents, which use:

- The same PUCT selection strategy as AlphaGo Zero [90], i.e. traversing the search tree by selecting actions a^* according to

$$a^* = \underset{a}{\operatorname{argmax}}\, \hat{Q}(s,a) + C_{puct}\pi(s,a)\frac{\sqrt{\sum_{a'} N(s,a')}}{1+N(s,a)},$$

  where $\hat{Q}(s,a)$ denotes the current estimated value (average backpropagated score) for action a in state s, C_{puct} denotes an exploration constant (set to $C_{puct} = 2.5$), $\pi(s,a)$ denotes the probability assigned to a in s by the trained policy, and $N(s,a)$ denotes the number of previous visits to the node that selecting a in s leads to in the search tree.
- An ϵ-greedy playout strategy where actions during playouts are sampled uniformly at random with probability $\epsilon = 0.5$, and sampled according to the trained policy π with probability $1-\epsilon = 0.5$.
- A straightforward expansion strategy that consists of expanding the tree by exactly one node per MCTS iteration.
- Values in the range $[-1, 1]$ for backpropagation; losses correspond to a value of -1, wins to a value of 1, draws to a value of 0, second place in a 6-player game to a value of $1 - \left((2-1) \times \frac{2}{6-1}\right) = 0.6$, etc. Note that this range of values is twice as big as the $[0, 1]$ range commonly used by other programs. To compensate for this, we set the $C_{puct} = 2.5$ exploration constant to a value that is approximately twice as large as the value used by programs such as AlphaZero (C_{puct} slowly shrinking from 1.25) [89] or ELF OpenGo ($C_{puct} = 1.5$) [105].
- Value estimates $\hat{Q}(s,a)$ equal to the parent node's value estimate for nodes that have no visits (i.e., $N(s,a) = 0$).
- Tree reuse, meaning that relevant parts of the search tree from previous search processes (in previous turns) are preserved for subsequent searches by the same agent [74,97,83].
- In stochastic games, an open-loop approach where nodes (other than the root node) do not store copies of game states, but only represent and collect statistics for the trajectories of actions leading up to them from the root node [75].

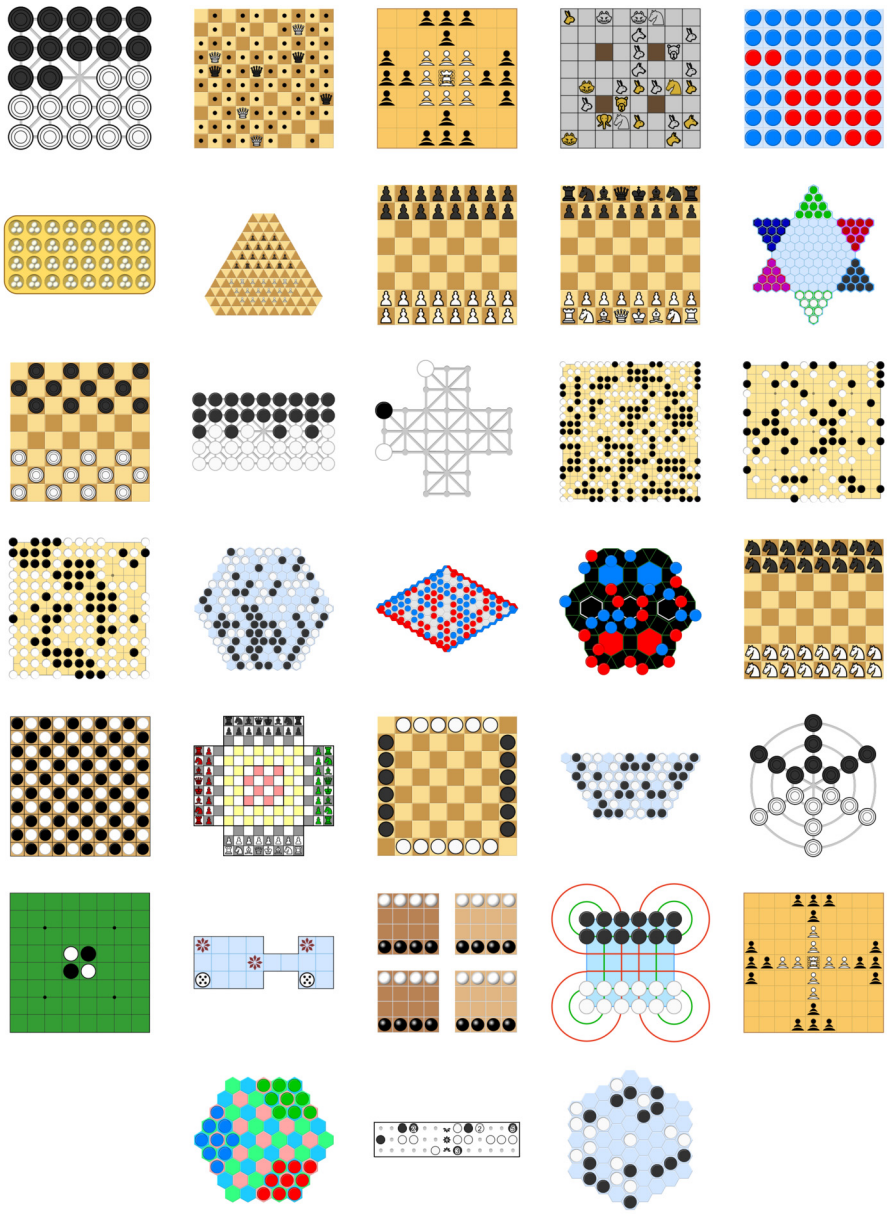


Fig. A.22. Screenshots of games used in experiments, generated by Ludii.

States are re-generated by applying those sequences of actions to copies of the root state as required in different MCTS iterations.

Every move, the MCTS agent corresponding to the player to move uses 1 second of thinking time, after which it selects a move proportionally to the distribution of visit counts among the children of the root node.

After every full game of self-play, we store collected samples of experience in replay buffers, where separate replay buffers per player each contain samples corresponding to game states in which the corresponding player is the player to move. We use Prioritized Experience Replay (PER) [84,96], with hyperparameters $\alpha = \beta = 0.5$. Each replay buffer has a maximum capacity of 2500 tuples of experience.

After every action in self-play, we run, for every player, one step of gradient descent to minimise the cross-entropy loss function (given for a single state s, representing a tuple of experience)

$$\mathcal{L}(s) = \left(-\boldsymbol{\pi}_{mcts}(s)^{\top} \log \boldsymbol{\pi}(s)\right),$$

where $\boldsymbol{\pi}_{mcts}(s)$ denotes the probability distribution over actions legal in s proportional to the visit counts resulting from an MCTS search in s, and $\boldsymbol{\pi}(s)$ similarly denotes the distribution over actions in the state s for the trained policy. We use a batch size of $N = 30$. Tuples of experience are additionally weighted based on the durations of the episodes from which they originate [96]. Weighted importance sampling is used to correct for PER (partially) and the weighting according to episode durations. Gradient descent steps are taken using a centred variant of RMSProp [44], with a base learning rate of 0.005, a momentum of 0.9, a discounting factor of 0.9, and a constant of 10^{-8} added to the denominator for stability. Additionally, we regularise by applying weight decay with $\lambda = 10^{-6}$ after every step.

After every game of self-play, we grow every player's feature set by up to two features, which are constructed by recombining two existing feature instances. We aim to construct new features such that the absolute correlation between its activity and the activity of either of its constituents is minimised, while the absolute correlation between its activity and the cross-entropy loss function is maximised [95]. Correlations for potential candidates are estimated from batches of size $N = 30$ sampled uniformly from the replay buffer. Every time, we generate at most one new proactive feature and one new reactive feature. Sometimes, either one or both of these may fail if all candidates turn out to be equivalent to a feature that already exists.

References

[1] R. Agarwal, M. Schwarzer, P.S. Castro, A. Courville, M.G. Bellemare, Deep reinforcement learning at the edge of the statistical precipice, in: Advances in Neural Information Processing Systems, 2021.
[2] T. Anthony, Z. Tian, D. Barber, Thinking fast and slow with deep learning and tree search, in: I. Guyon, U.V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, R. Garnett (Eds.), Advances in Neural Information Processing Systems, vol. 30, Curran Associates, Inc., 2017, pp. 5360–5370.
[3] N. Araki, K. Yoshida, Y. Tsuruoka, J. Tsujii, Move prediction in Go with the maximum entropy method, in: Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games, IEEE, 2007, pp. 189–195.
[4] P. Auer, N. Cesa-Bianchi, P. Fischer, Finite-time analysis of the multiarmed bandit problem, Mach. Learn. 47 (2002) 235–256.
[5] D. Beal, M. Clarke, The construction of economical and correct algorithms for king and pawn against king, in: M. Clarke (Ed.), Advances in Computer Chess 2, Edinburgh University Press, Edinburgh, 1980, pp. 1–30.
[6] X. Bouthillier, P. Delaunay, M. Bronzi, A. Trofimov, B. Nichyporuk, J. Szeto, N. Mohammadi Sepahvand, E. Raff, K. Madan, V. Voleti, S. Ebrahimi Kahou, V. Michalski, T. Arbel, C. Pal, G. Varoquaux, P. Vincent, Accounting for variance in machine learning benchmarks, in: A. Smola, A. Dimakis, I. Stoica (Eds.), Proceedings of Machine Learning and Systems, vol. 3, 2021, pp. 747–769.
[7] B. Bouzy, Associating domain-dependent knowledge and Monte Carlo approaches within a Go program, Inf. Sci. 175 (2005) 247–257.
[8] B. Bouzy, G. Chaslot, Bayesian generation and integration of K-nearest-neighbor patterns for 19x19 Go, in: G. Kendall, S. Lucas (Eds.), Proceedings of the 2005 IEEE Symposium on Computational Intelligence in Games, IEEE, 2005, pp. 176–181.
[9] I. Bratko, D. Kopec, D. Michie, Pattern-based representation of chess end-game knowledge, Comput. J. 21 (1978) 149–153.
[10] M.M. Bronstein, J. Bruna, T. Cohen, P. Veličković, Geometric deep learning: grids, groups, graphs, geodesics, and gauges, https://arxiv.org/abs/2104.13478, 2021.
[11] C. Browne, Hex Strategy: Making the Right Connections, AK Peters, Massachusetts, 2000.
[12] C. Browne, Modern techniques for ancient games, in: IEEE Conference on Computational Intelligence and Games, IEEE Press, Maastricht, 2018, pp. 490–497.
[13] C. Browne, É. Piette, M. Stephenson, D.J.N.J. Soemers, General board geometry, in: C. Browne, A. Kishimoto, J. Schaeffer (Eds.), Advances in Computer Games (ACG 2021), in: Lecture Notes in Computer Science, vol. 13262, Springer, Cham, 2022, pp. 235–246.
[14] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P.I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, S. Colton, A survey of Monte Carlo tree search methods, IEEE Trans. Comput. Intell. AI Games 4 (2012) 1–49.
[15] C. Browne, D.J.N.J. Soemers, E. Piette, Strategic features for general games, in: Proceedings of the 2nd Workshop on Knowledge Extraction from Games (KEG), 2019, pp. 70–75.
[16] C. Browne, D.J.N.J. Soemers, É. Piette, M. Stephenson, M. Conrad, W. Crist, T. Depaulis, E. Duggan, F. Horn, S. Kelk, S.M. Lucas, J.P. Neto, D. Parlett, A. Saffidine, U. Schädler, J.N. Silva, A. de Voogt, M.H.M. Winands, Foundations of Digital Archæoludology, Technical Report, Schloss Dagstuhl Research Meeting, Germany, 2019.
[17] C. Browne, M. Stephenson, É. Piette, D.J.N.J. Soemers, A practical introduction to the Ludii general game system, in: T. Cazenave, H.J. van den Herik, A. Saffidine, I.-C. Wu (Eds.), Advances in Computer Games. ACG 2019, in: Lecture Notes in Computer Science, vol. 12516, Springer, Cham, 2020, pp. 167–179.
[18] C.B. Browne, Connection Games: Variations on a Theme, AK Peters, Massachusetts, USA, 2005.
[19] C.B. Browne, Automatic Generation and Evaluation of Recombination Games, Phd thesis, Faculty of Information Technology, Queensland University of Technology Queensland, Australia, 2009.
[20] M. Buro, From simple features to sophisticated evaluation functions, in: First International Conference on Computers and Games, 1999, pp. 126–145.
[21] M. Campbell, A.J. Hoane Jr., F. Hsu, Deep blue, Artif. Intell. 134 (2002) 57–83.
[22] J. Carr, Using graph convolutional networks and td(λ) to play the game of risk, https://arxiv.org/abs/2009.06355, 2020.
[23] T. Cazenave, Automatic acquisition of tactical Go rules, in: Proceedings of the Game Programming Workshop in Japan'96, 1996.
[24] T. Cazenave, Mobile networks for computer Go, IEEE Trans. Games 14 (2022) 76–84.
[25] T. Cazenave, Y.-C. Chen, G. Chen, S.-Y. Chen, X.-D. Chiu, J. Dehos, M. Elsa, Q. Gong, H. Hu, V. Khalidov, C.-L. Li, H.-I. Lin, Y.-J. Lin, X. Martinet, V. Mella, J. Rapin, B. Roziere, G. Synnaeve, F. Teytaud, O. Teytaud, S.-C. Ye, Y.-J. Ye, S.-J. Yen, S. Zagoruyko, Polygames: improved zero learning, ICGA J. 42 (2020) 244–256.
[26] V. Chvatal, A greedy heuristic for the set-covering problem, Math. Oper. Res. 4 (1979) 233–235.
[27] J. Clune, Heuristic evaluation functions for general game playing, in: Proceedings of the 22nd National Conference on Artificial Intelligence, AAAI Press, 2007, pp. 1134–1139.
[28] T.S. Cohen, Equivariant Convolutional Networks, Ph.D. thesis, University of Amsterdam, Amsterdam, the Netherlands, 2021.
[29] Q. Cohen-Solal, Learning to play two-player perfect-information games without knowledge, https://arxiv.org/abs/2008.01188, 2020.
[30] Q. Cohen-Solal, T. Cazenave, Minimax strikes back, in: AAAI-21 Workshop on Reinforcement Learning in Games, 2021.
[31] R. Coulom, Computing "Elo ratings" of move patterns in the game of Go, ICGA J. 30 (2007) 198–208.
[32] R. Coulom, Efficient selectivity and backup operators in Monte-Carlo tree search, in: H.J. van den Herik, P. Ciancarini, H.H.L.M. Donkers (Eds.), Computers and Games, in: Lecture Notes in Computer Science, vol. 4630, Springer, Berlin, Heidelberg, 2007, pp. 72–83.

[33] E. Cox, E. Schkufza, R. Madsen, M.R. Genesereth, Factoring general games using propositional automata, in: Y. Björnsson, P. Stone, M. Thielscher (Eds.), IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA), 2009, pp. 13–20.
[34] M. Davis, G. Logemann, D. Loveland, A machine program for theorem proving, Commun. ACM 5 (1962) 394–397.
[35] M. Davis, H. Putnam, A computing procedure for quantification theory, J. ACM 7 (1960) 201–215.
[36] S. Draper, Propnets - moving to an external representation, http://sanchoggp.blogspot.com/2014/05/propnets-moving-to-external.html, 2014.
[37] R. Dror, S. Shlomov, R. Reichart, Deep dominance - how to properly compare deep neural models, in: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, 2019, pp. 2773–2785.
[38] H.D. Enderton, The Golem Go Program, Technical Report CMU-CS-92-101, School of Computer Science, Carnegie-Mellon University, 1991.
[39] H. Finnsson, Y. Björnsson, Learning simulation control in general game-playing agents, in: Proceedings of the 24th AAAI Conference on Artificial Intelligence, AAAI Press, 2010, pp. 954–959.
[40] D. Fotland, Knowledge representation in The Many Faces of Go, http://www.smart-games.com/knowpap.txt, 1993.
[41] S. Gelly, D. Silver, Combining online and offline knowledge in UCT, in: Proceedings of the 24th International Conference on Machine Learning, 2007, pp. 273–280.
[42] S. Gelly, Y. Wang, R. Munos, O. Teytaud, Modification of UCT with Patterns in Monte-Carlo Go, Technical Report RR-6062, INRIA, Paris, 2006.
[43] M.R. Genesereth, Y. Björnsson, The international general game playing competition, AI Mag. 34 (2013) 107–111.
[44] A. Graves, Generating sequences with recurrent neural networks, https://arxiv.org/abs/1308.0850v5, 2013.
[45] S.-C. Huang, B. Arneson, R.B. Hayward, M. Müller, J. Pawlewicz, Mohex 2.0: a pattern-based MCTS Hex player, in: H. van den Herik, H. Iida, A. Plaat (Eds.), Computers and Games. CG 2013, in: Lecture Notes in Computer Science, vol. 8427, Springer, Cham, 2014, pp. 60–71.
[46] J. Janisch, T. Pevný, V. Lisý, Classification with costly features as a sequential decision-making problem, Mach. Learn. 109 (2020) 1587–1615.
[47] R.G. Jeroslow, J. Wang, Solving propositional satisfiability problems, Ann. Math. Artif. Intell. 1 (1990) 167–187.
[48] R.M. Karp, Reducibility among combinatorial problems, in: R.E. Miller, J.W. Thatcher, J.D. Bohlinger (Eds.), Complexity of Computer Computations the IBM Research Symposia Series, Plenum Press, 1972, pp. 85–103.
[49] M. Kirci, N. Sturtevant, J. Schaeffer, A GGP feature learning algorithm, Künstl. Intell. 25 (2011) 35–42.
[50] L. Kocsis, C. Szepesvári, Bandit based Monte-Carlo planning, in: J. Fürnkranz, T. Scheffer, M. Spiliopoulou (Eds.), Machine Learning: ECML 2006, in: Lecture Notes in Computer Science, vol. 4212, Springer, Berlin, Heidelberg, 2006, pp. 282–293.
[51] F. Koriche, S. Lagrue, É. Piette, S. Tabary, Constraint-based symmetry detection in general game playing, in: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17, 2017, pp. 280–287.
[52] G. Kuhlmann, K. Dresner, P. Stone, Automatic heuristic construction in a complete general game player, in: Proceedings of the 21st National Conference on Artificial Intelligence, AAAI Press, 2006, pp. 1457–1462.
[53] Y. LeCun, Y. Bengio, G. Hinton, Deep learning, Nature 521 (2015) 436–444.
[54] Y. LeCun, B. Boser, J.S. Denker, D. Henderson, R.E. Howard, W. Hubbard, L.D. Jackel, Backpropagation applied to handwritten zip code recognition, Neural Comput. 1 (1989) 541–551.
[55] R. Levinson, R. Snyder, Adaptive pattern-oriented chess, in: Proceedings of the 9th National Conference on Artificial Intelligence, in: AAAI, vol. 2, 1991, pp. 601–605.
[56] H. Levy, Stochastic dominance and expected utility: survey and analysis, Manag. Sci. 38 (1992).
[57] C. Linnæus, Iter Lapponicum, Stockholm, Sweden, 1732.
[58] R.J. Lorentz, T.E. Zosa IV, Machine learning in the game of Breakthrough, in: M.H.M. Winands, H. van den Herik, W.A. Kosters (Eds.), Advances in Computer Games, in: Lecture Notes in Computer Science, vol. 10664, Springer, 2017, pp. 140–150.
[59] N. Love, T. Hinrichs, D. Haley, E. Schkufza, M. Genesereth, General Game Playing: Game Description Language Specification, Technical Report LG-2006-01, Stanford Logic Group, 2008.
[60] É. Lucas, Récréations scientifiques, Nature (1887) 402–404.
[61] J. Mańdziuk, Towards cognitively-plausible game playing systems, IEEE Comput. Intell. Mag. 6 (2011) 38–51.
[62] H.B. Mann, D.R. Whitney, On a test of whether one of two random variables is stochastically larger than the other, Ann. Math. Stat. 18 (1947) 50–60.
[63] M.V. Mannino, V.S. Mookerjee, Optimizing expert systems: heuristics for efficiently generating low-cost information acquisition strategies, INFORMS J. Comput. 11 (1999) 278–291.
[64] D. Michulke, Heuristic interpretation of predicate logic expressions in general game playing, in: Proceedings of the IJCAI-11 Workshop on General Game Playing (GIGA '11), 2011, pp. 61–68.
[65] D. Michulke, S. Schiffel, Distance features for general game playing agents, in: Proceedings of the 4th International Conference on Agents and Artificial Intelligence, 2012, pp. 127–136.
[66] D. Michulke, S. Schiffel, Admissible distance heuristics for general games, in: J. Filipe, A. Fred (Eds.), Agents and Artificial Intellgence, ICAART 2012, in: Communications in Computer and Information Science, vol. 358, Springer, Berlin, Heidelberg, 2013.
[67] F. Morandin, G. Amato, R. Gini, C. Metta, M. Parton, G.-C. Pascutto, SAI: a sensible artificial intelligence that plays Go, in: Proceedings of the 2019 International Joint Conference on Neural Networks (IJCNN), IEEE, 2019.
[68] M. Müller, Pattern matching in explorer, in: Proceedings of the Game Playing System Workshop, 1991, pp. 1–3.
[69] M. Müller, Computer Go as a Sum of Local Games: an Application of Combinatorial Game Theory, Phd thesis, Swiss Federal Institute of Technology Zürich, Zürich, Switzerland, 1995.
[70] M. Müller, Computer Go, Artif. Intell. 134 (2002) 145–179.
[71] H.J.R. Murray, A History of Board-Games Other than Chess, Clarendon Press, Oxford, United Kingdom, 1951.
[72] F. Nan, J. Wang, V. Saligrama, Feature-budgeted random forest, in: Proceedings of the 32nd International Conference on Machine Learning, in: PMLR, vol. 37, 2015, pp. 1983–1991.
[73] F. Nan, J. Wang, V. Saligrama, Pruning random forests for prediction on a budget, in: D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, R. Garnett (Eds.), Advances in Neural Information Processing Systems 29 (NIPS 2016), Curran Associates, Inc., 2016, pp. 2334–2342.
[74] T. Pepels, M.H.M. Winands, M. Lanctot, Real-time Monte Carlo tree search in Ms Pac-Man, IEEE Trans. Comput. Intell. AI Games 6 (2014) 245–257.
[75] D. Perez, J. Dieskau, M. Hünermund, S. Mostaghim, S.M. Lucas, Open loop search for general video game playing, in: Proceedings of the Genetic and Evolutionary Computation Conference, ACM, 2015, pp. 337–344.
[76] É. Piette, C. Browne, D.J.N.J. Soemers, Ludii game logic guide, https://arxiv.org/abs/2101.02120, 2021.
[77] É. Piette, D.J.N.J. Soemers, M. Stephenson, C.F. Sironi, M.H.M. Winands, C. Browne, Ludii – the ludemic general game system, in: G.D. Giacomo, A. Catala, B. Dilkina, M. Milano, S. Barro, A. Bugarín, J. Lang (Eds.), Proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020), in: Frontiers in Artificial Intelligence and Applications, vol. 325, IOS Press, 2020, pp. 411–418.
[78] É. Piette, M. Stephenson, D.J.N.J. Soemers, C. Browne, General board game concepts, in: Proceedings of the 2021 IEEE Conference on Games (CoG), IEEE, 2021, pp. 932–939.
[79] J. Pitrat, Realization of a general game-playing program, in: IFIP Congress (2), 1968, pp. 1570–1574.
[80] D. Pretolani, Efficiency and stability of hypergraph SAT algorithms, in: DIMACS Workshop: Cliques, Coloring, and Satisfiability 1993, 1993, pp. 479–498.
[81] T. Raiko, J. Peltonen, Application of UCT search to the connection games of Hex, Y, *Star, and Renkula!, in: Proceedings of the Finnish Artificial Intelligence Conference, 2008, pp. 89–93.

[82] A. Saffidine, N. Jouandeau, T. Cazenave, Solving breakthrough with race patterns and job-level proof number search, in: Advances in Computer Games (ACG 2011), in: Lecture Notes in Computer Science, vol. 7168, Springer, Berlin, Heidelberg, 2012, pp. 196–207.
[83] A.M.L. Santos, Monte Carlo Tree Search Experiments in Hearthstone, Masters thesis, Técnico Lisboa, Lisbon, Portugal, 2017.
[84] T. Schaul, J. Quan, I. Antonoglou, D. Silver, Prioritized experience replay, in: International Conference on Learning Representations (ICLR), 2016.
[85] S. Schiffel, M. Thielscher, Fluxplayer: a successful general game player, in: Proceedings of the 22nd National Conference on Artificial Intelligence, AAAI Press, 2007, pp. 1191–1196.
[86] E. Schkufza, N. Love, M.R. Genesereth, Propositional automata and cell automata: representational frameworks for discrete dynamic systems, in: AI 2008: Advances in Artificial Intelligence, vol. 5360, 2008, pp. 56–66.
[87] E. Shelhamer, J. Long, T. Darrell, Fully convolutional networks for semantic segmentation, IEEE Trans. Pattern Anal. Mach. Intell. 39 (2017) 640–651.
[88] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, D. Hassabis, Mastering the game of Go with deep neural networks and tree search, Nature 529 (2016) 484–489.
[89] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, D. Hassabis, A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play, Science 362 (2018) 1140–1144.
[90] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, D. Hassabis, Mastering the game of Go without human knowledge, Nature 550 (2017) 354–359.
[91] D. Silver, R. Sutton, M. Müller, Reinforcement learning of local shape in the game of Go, in: Proceedings of the 20th International Joint Conference on Artificial Intelligence, 2007, pp. 1053–1058.
[92] P. Skowronski, Y. Björnsson, M.H.M. Winands, Automated discovery of search-extension features, in: H.J. van den Herik, P. Spronck (Eds.), Advances in Computer Games, in: Lecture Notes in Computer Science, vol. 6048, Springer, Berlin, Heidelberg, 2009.
[93] D.J.N.J. Soemers, V. Mella, C. Browne, O. Teytaud, Deep learning for general game playing with Ludii and Polygames, ICGA J. 43 (2022) 146–161.
[94] D.J.N.J. Soemers, V. Mella, É. Piette, M. Stephenson, C. Browne, O. Teytaud, Transfer of fully convolutional policy-value networks between games and game variants, https://arxiv.org/abs/2102.12375, 2021.
[95] D.J.N.J. Soemers, É. Piette, C. Browne, Biasing MCTS with features for general games, in: Proceedings of the 2019 IEEE Congress on Evolutionary Computation, IEEE, 2019, pp. 442–449.
[96] D.J.N.J. Soemers, É. Piette, M. Stephenson, C. Browne, Manipulating the distributions of experience used for self-play learning in Expert Iteration, in: Proceedings of the 2020 IEEE Conference on Games, IEEE, 2020, pp. 245–252.
[97] D.J.N.J. Soemers, C.F. Sironi, T. Schuster, M.H.M. Winands, Enhancements for real-time Monte-Carlo tree search in general video game playing, in: Proceedings of the 2016 IEEE Conference on Computational Intelligence in Games, IEEE, 2016, pp. 436–443.
[98] M. Stephenson, W. Crist, C. Browne, Digital Ludeme Project Database Guide, Technical Report, Maastricht University, 2020, https://ludii.games/downloads/DLP_Database_Guide.pdf.
[99] D. Stern, R. Herbrich, T. Graepel, Bayesian pattern ranking for move prediction in the game of Go, in: W.W. Cohen, A. Moore (Eds.), Proceedings of the 23rd International Conference on Machine Learning, 2006, pp. 873–880.
[100] D. Stoutamire, Machine Learning, Game Play, and Go, Technical Report TR 91-128, Center for Automation and Intelligent Systems Research, Case Western Reserve University, 1991.
[101] N.R. Sturtevant, A.M. White, Feature construction for reinforcement learning in Hearts, in: H.J. van den Herik, P. Ciancarini, H.H.L.M. Donkers (Eds.), Computers and Games, in: Lecture Notes in Computer Science, vol. 4630, Springer, 2007, pp. 122–134.
[102] R.S. Sutton, A.G. Barto, Reinforcement Learning: An Introduction, 2nd ed., MIT Press, Cambridge, MA, 2018.
[103] M. Świechowski, H. Park, J. Mańdziuk, K.-J. Kim, Recent advances in general game playing, Sci. World J. 2015 (2015).
[104] O. Syed, A. Syed, Arimaa - a new game designed to be difficult for computers, ICGA J. 26 (2003) 138–139.
[105] Y. Tian, J. Ma, Q. Gong, S. Sengupta, Z. Chen, J. Pinkerton, C.L. Zitnick, ELF OpenGo: an analysis and open reimplementation of AlphaZero, in: Proceedings of the 36th International Conference on Machine Learning (ICML), 2019, pp. 6244–6253.
[106] K. Trapeznikov, V. Saligrama, Supervised sequential classification under budget constraints, in: Proceedings of the 16th International Conference on Artificial Intelligence and Statistics (AISTATS), in: PMLR, vol. 31, 2013, pp. 581–589.
[107] T. Urvoy, gnugo team, Pattern matching in Go with DFA, http://tanguy.urvoy.free.fr/Papers/dfabstract.pdf, 2002.
[108] P.E. Utgoff, D. Precup, Constructive function approximation, Feature Extraction, Construction, and Selection: A Data-Mining Perspective 453 (1998) 219–235.
[109] K. Walędzik, J. Mańdziuk, Multigame playing by means of UCT enhanced with automatically generated evaluation functions, in: Artificial General Intelligence: 4th International Conference, in: Lecture Notes in Computer Science, vol. 6830, Springer, 2011, pp. 327–332.
[110] K. Walędzik, J. Mańdziuk, An automatically generated evaluation function in general game playing, IEEE Trans. Comput. Intell. AI Games 6 (2014) 258–270.
[111] E.C.D. van der Werf, J.W.H.M. Uiterwijk, E.O. Postma, H.J. van den Herik, Local move prediction in Go, in: J. Schaeffer, M. Müller, Y. Björnsson (Eds.), Computers and Games, in: Lecture Notes in Computer Science, vol. 2883, Springer, 2003.
[112] D.J. Wu, Accelerating self-play learning in Go, arXiv:1902.10565v3, 2019.
[113] A.L. Zobrist, A New Hashing Method with Application for Game Playing, Technical Report 88, Computer Sciences Deptartment, the University of Wisconsin, Madison, 1970, Republished in ICCA Journal in 1990.