Split Moves for Monte-Carlo Tree Search

Jakub Kowalski[1], **Maksymilian Mika**[1], **Wojciech Pawlik**[1], **Jakub Sutowicz**[1], **Marek Szykuła**[1], **Mark H. M. Winands**[2]

[1]University of Wrocław, Faculty of Mathematics and Computer Science
[2]Maastricht University, Department of Data Science and Knowledge Engineering
jko@cs.uni.wroc.pl, mika.maksymilian@gmail.com, pawlik.wj@gmail.com,
jakubsutowicz@gmail.com, msz@cs.uni.wroc.pl, m.winands@maastrichtuniversity.nl

Abstract

In many games, moves consist of several decisions made by the player. These decisions can be viewed as separate moves, which is already a common practice in multi-action games for efficiency reasons. Such division of a player move into a sequence of simpler / lower level moves is called *splitting*. So far, split moves have been applied only in forementioned straightforward cases, and furthermore, there was almost no study revealing its impact on agents' playing strength. Taking the knowledge-free perspective, we aim to answer how to effectively use split moves within Monte-Carlo Tree Search (MCTS) and what is the practical impact of split design on agents' strength. This paper proposes a generalization of MCTS that works with arbitrarily split moves. We design several variations of the algorithm and try to measure the impact of split moves separately on efficiency, quality of MCTS, simulations, and action-based heuristics. The tests are carried out on a set of board games and performed using the Regular Boardgames General Game Playing formalism, where split strategies of different granularity can be automatically derived based on an abstract description of the game. The results give an overview of the behavior of agents using split design in different ways. We conclude that split design can be greatly beneficial for single- as well as multi-action games.

Introduction

The benefits of simulation-based, knowledge-free, open-loop algorithms such as Monte-Carlo Tree Search (MCTS) (Kocsis and Szepesvári 2006; Browne et al. 2012; Świechowski et al. 2021) and Rolling Horizon Evolutionary Algorithm (Perez et al. 2013) are especially suited to work within environments with many unknowns. In particular, they are widely used in General Game Playing (GGP) (Genesereth, Love, and Pell 2005), a domain focusing on developing agents that can successfully play any game given its formalized rules, which was established to promote work in generalized, practically applicable algorithms (Finnsson and Björnsson 2010; Goldwaser and Thielscher 2020). Initially based entirely on Stanford's Game Description Language (GDL) (Love et al. 2006), GGP expands over time as new game description formalisms are being developed e.g., Toss (Kaiser and Stafiniak 2011), GVG-AI (Perez et al. 2016), Regular Boardgames (RBG) (Kowalski et al. 2019), and Ludii (Piette et al. 2020).

Recent advances in search and learning support the trend of generalization, focusing on methods being as widely applicable as possible. Deep Q-networks were applied to play classic Atari games and achieved above human-level performance on most of the 49 games from the test set (Mnih et al. 2015). More recently, AlphaZero, showed how to utilize a single technique to play Go, Chess, and Shogi on a level above all other compared AI agents (Silver et al. 2018).

In the trend of developing enhancements for MCTS (Cazenave 2015; Baier and Winands 2018), we tackle the problem of influencing the quality of the search by altering the structure of the game tree itself. In many games, a player's turn consists of a sequence of choices that can be examined separately. A straightforward representation is to encode these choices as distinct moves, obtaining a split game tree, instead of using a single move in orthodox design. The potential applications go beyond games, as the method can be used for any kind of problem that is solvable via a simulation-based approach and its representation of actions can be decomposed. In this paper, we are interested in the general technique of altering the game tree by introducing split moves and its possible effects, rather than its application to a particular game combined with expert knowledge. Hence, we focus on the GGP setting and MCTS, which is the most well-known and widely applied general search algorithm working without expert knowledge.

We propose the *semisplit* algorithm, which is a generalization of MCTS that effectively works with arbitrary splitting. For the purposes of experiments, we implement the concept in the Regular Boardgames system (Kowalski et al. 2019) – a universal GGP formalism for the class of finite deterministic games with perfect information. A few *split strategies* of different granularity are developed, which split moves basing on the given general game description. The experiments are conducted on a set of classic board games, comparing agents using orthodox and split designs. We test a number of variants of the algorithm, applying split moves selectively to different phases of MCTS, and include action-based heuristics (MAST, RAVE (Gelly and Silver 2007)) to observe the behavior also for enhanced MCTS. From the results, we identify the most beneficial configurations and conclude that split moves can greatly improve the player's strength.

The full version of this paper is available at (Kowalski et al. 2021b), and the source code used for the experiments is shared within the RBG implementation (Kowalski et al. 2021a).

Related Work

The idea of splitting moves is well known, but apparently, it was not given proper consideration in the literature, being either used naturally in trivial cases or restrained to follow a human-authored heuristic. Even these cases were discussed in rather limited aspects, given how general applications of split technique can be.

For particularly complex environments, split is regarded as natural and mandatory. This technique is widely used for Arimaa, Hearthstone, and other multi-action games (Fotland 2006; Justesen et al. 2017; Roelofs 2017). Here, the reduced branching factor is considered to be the main effect, as otherwise, programs could not play such games at a proper level. The case of Amazons is the only one that we have found where agents playing with split and non-split representations were compared against each other (Kloetzer, Iida, and Bouzy 2007). For multi-action games such as real-time strategies, where the ordering of actions is unrestricted, Combinatorial Multi-armed Bandits algorithms are often employed (Ontañón 2017). Using actions separately as moves in the MCTS tree was also considered under the name of *hierarchical expansion* (Roelofs 2017) and in context of factoring action space for MDPs (Geißer, Speck, and Keller 2020). So far, splitting was applied only for such multi-action games, where it is possible and natural to divide a turn into separate moves and process them like regular ones. Splitting / move decomposition should not be confused with the game decomposition (Hufschmitt, Vittaut, and Jouandeau 2019).

A practical application of splitting is found in GGP, where many games are manually (re)encoded in their split variants just to improve efficiency. For example, some split versions of games like Amazons, Arimaa, variants of Draughts, or Pentago exist in GDL. The same approach is taken in other systems like Ludii. However, it is generally unknown how such versions affect the agents' playing strength.

Additionally, splitting via artificial turns causes some repercussions, e.g., for game statistics or handling the turn timer. Especially in a competitive setting, an agent gets the same time for every move, even when they are single actions, thus making more moves in a turn gives longer computation time. Although this can be resolved, it requires specific language constructions that do not exist in any GGP system. Instead of complicating languages, it would be better and more general to develop split-handling on the agent's side.

A related topic is *move groups* (Saito et al. 2007; Childs, Brodeur, and Kocsis 2008; Van Eyck and Müller 2011), where during MCTS expansion, children of every tree node are partitioned into a constant number of classes guided by a heuristic. The basic idea of move groups is to divide nodes of the MCTS tree into two levels, creating intermediate nodes that group children belonging to the same class. There is no reported application of move groups beyond Go, Settlers of Catan, and artificial single-player game trees for maximizing UCT payoff. Usually, move groups are understood as introducing artificial tree nodes unrelated to any move representation nor simplifying computation. They also require human intervention to encode rules on how to partition the moves. From the perspective of splitting, (nested) move groups are a side effect, but not every partition can be obtained by splitting.

As splitting just alters the game tree, it is compatible with any other search algorithm that uses this game tree as the underlying structure. In particular, it affects action-based heuristics operating on moves such as MAST and RAVE (Gelly and Silver 2007). But unlike usual techniques, splitting often improves efficiency, so it would be beneficial assuming that it leaves the behavior of algorithms unchanged.

Semisplitting in MCTS

Abstract Game

We adapt a standard definition of an abstract turn-based game (Rasmusen 2007) to our goals. A *finite deterministic turn-based game with perfect information* (simply called *game*) $\mathcal{G}$ is a tuple $(players_{\mathcal{G}}, \mathcal{T}_{\mathcal{G}}, control_{\mathcal{G}}, out_{\mathcal{G}})$, where: $players_{\mathcal{G}}$ is a finite non-empty set of *players*; $\mathcal{T}_{\mathcal{G}} = (V, E)$ is a finite directed tree called the *game tree*, where V is the set of nodes called *game states* and E is the set of edges called *moves*, $V_{\mathrm{n}} \subset V$ is the set of inner nodes called *non-terminal* states, and $V_{\mathrm{t}} \subseteq V$ is the set of leaves called *terminal* states; $control_{\mathcal{G}} \colon V_{\mathrm{n}} \to players_{\mathcal{G}}$ is a function assigning the *current player* to non-terminal states; $out_{\mathcal{G}} \colon V_{\mathrm{t}} \times players_{\mathcal{G}} \to \mathbb{R}$ is a function assigning the final *score* of each player at terminal states. For a non-terminal state $s \in V_{\mathrm{n}}$, the set of *legal moves* is the set of outgoing edges $\{(s,t) \in E \mid t \in V\}$. During a play, the current player $control_{\mathcal{G}}(s)$ chooses one of its legal moves. The game tree is directed outwards its root $s_0 \in V$, which is called the *initial state*. Hence, all states are reachable from s_0. Each play starts from s_0 and ends at a terminal state (leaf).

Two games are *isomorphic* if there exists a bijection between the states that preserve the edges of the game tree and the current player (if the state is non-terminal) or the scores (if the state is terminal). For a state $s \in V$, the *subgame* $\mathcal{G}_s$ is the game with the tree obtained from $\mathcal{T}_{\mathcal{G}}$ by rooting at s and removing all states unreachable from s.

Semisplit Game

Going deeper into a particular representation of a move, it usually can be partitioned into a sequence of smaller pieces, which we call *semimoves*. For example, depending on a particular implementation, they could correspond to atomic actions, move groups, or, in the extreme case, even single bits of a technical move representation. Computing semimoves can be, but not always is, computationally easier than full moves and sometimes may reveal structural information desirable in a knowledge-based analysis.

However, often a natural and the most effective splitting does not lead to a proper variant of the game, i.e., we cannot treat semimoves as usual moves. It is because not every available sequence of easily computed semimoves can be completed up to a legal move. This especially concerns

single-action games, but also splits inferred automatically in general, where without prior knowledge it is difficult to determine if we obtain a proper game.

Example 1 *In Chess, a typical move consists of picking up a piece and choosing its destination square. It would be much more efficient first to select a piece from the list of pieces and then a square from the list of available destinations computed just for this piece, than to select a move from the list of all legal ones, which is usually much longer. However, sometimes we may not be able to make a legal move with a selected piece, e.g., because the piece is blocked or the king will be left under check.*

A remedy could be checking if each available semimove is a prefix of at least one legal move. However, in many cases, this can be as costly as computing all legal moves, losing performance benefits, or even decreasing efficiency. Instead, we can work on semisplit games directly.

To provide an abstract model, we require a different game definition, including additional information about intermediate states. We extend the definition of a game to a *semisplit game* as follows. Let V be now the disjoint union of non-terminal states V_{n}, terminal states V_{t}, and the *intermediate states* V_{i}. Then, V_{n} is a subset of inner vertices of the game tree, terminal states V_{t} is a subset of leaves, and V_{i} can contain states of both types. The states in V_{n} and V_{t} are called *nodal*. A semisplit game must satisfy that the initial state s_0 is nodal, and for every non-terminal state $s \in V_{\mathrm{n}}$, the subgame $\mathcal{G}_s$ contains at least one terminal state. The second condition ensures that from each nodal state, a terminal state is reachable. Yet, for an intermediate state, there may be no nodal state in its subgame; then this state is called *dead*. An edge is now called a *semimove*. A *submove* is a directed path where nodal states can occur only at the beginning or at the end, and there are only intermediate states in the middle. Then, a *move* is a submove between two nodal states.

There is a correspondence between a semisplit game and an (ordinary) game. The *rolled-up* game of a semisplit game is obtained by removing all dead states, and then by replacing each maximal connected component rooted at a non-terminal state with only intermediate states below with one non-terminal state; then all the edges become moves. A semisplit game $\mathcal{G}'$ is *equivalent* to a game $\mathcal{G}$ if the rolled-up game of $\mathcal{G}'$ is isomorphic to $\mathcal{G}$. So generally, splitting moves in a game means deriving its equivalent semisplit game.

An example of two distinct semisplit versions of Chess is shown in Fig. 1.

The Semisplit Algorithm

We describe a generalization of MCTS that works on an underlying semisplit game. To distinguish from the standard MCTS algorithm that operates on an ordinary game, the latter is called *orthodox MCTS*. In the following description, we use standard terminology (Browne et al. 2012) and focus on the differences with orthodox MCTS.

The simulation phase is shown in Alg. 1, lines 1–13. Drawing a move at random is realized through backtracking (SEMISPLITRANDOMMOVE). Given a game state, we choose and apply semimoves in the same way as moves in orthodox MCTS, but we keep the list of legal semimoves computed at each level. When it happens that the current intermediate state does not have a legal semimove (it is dead), we backtrack and try another semimove. Thus, a move is always found if it exists, and every legal move has a positive chance to be chosen, although the probability distribution (on legal moves) may be not uniform (in the worst case, the probability of choosing a move can be exponentially smaller in the number of introduced intermediate states). A single simulation (SEMISPLITSIMULATION) just uses the modified random move selection. Note that this function can fail (line 4), which happens if and only if called for a dead state.

The vanilla variant of semisplit MCTS is shown in Alg. 1. As orthodox MCTS, it uses the UCT policy in the selection phase (Kocsis and Szepesvári 2006). But in contrast, semis-

Algorithm 1: Vanilla semisplit MCTS.

```
Input: s – game state
 1: function SemisplitSimulation(s)
 2:     while not s.IsTerminal() do
 3:         m ← SemisplitRandomMove(s)
 4:         if m = None then return None                ▷ s is dead
 5:         s ← s.Apply(m)
 6:     return s.Scores()
 7: function SemisplitRandomMove(s)
 8:     for all a ∈ Shuffle(s.GetAllSemimoves()) do
 9:         s' ← s.apply(a)
10:         if s'.IsNodal() then return a
11:         m ← SemisplitRandomMove(s')
12:         if m ≠ None then return Concatenate(a, m)
13:     return None                                     ▷ No legal move
14: function MCTSIteration( )
15:     v ← TreeRoot()
16:     while not v.State().IsTerminal() do
17:         if v.FullyExpanded() then
18:             v ← v.UCT()
19:         else
20:             a_1 ← RandomElement(v.UntriedSemimoves())
21:             (v', scores) ← Expand(v, a_1)
22:             if scores = None then continue
23:             Backpropagation(v', scores)
24:             return
25:     Backpropagation(v, v.State().Scores())
Input: v – leaf node; a_1 – selected untried semimove
26: function Expand(v, a_1)
27:     s ← v.State().Apply(a_1)
28:     scores ← SemisplitSimulation(s)
29:     if scores = None then
30:         v.RemoveSemimove(a_1)
31:         return None
32:     c ← CreateNode(s)
33:     v.AddChild(c, a_1)
34:     return (c, scores)
Input: v – leaf node; scores – players' final scores
35: function Backpropagation(v, scores)
36:     while v ≠ TreeRoot() do
37:         v.scoreSum ← v.scoreSum + scores[v.Player]
38:         v.iterations ← v.iterations + 1
39:         v ← v.Parent()
40:     v.iterations ← v.iterations + 1
```

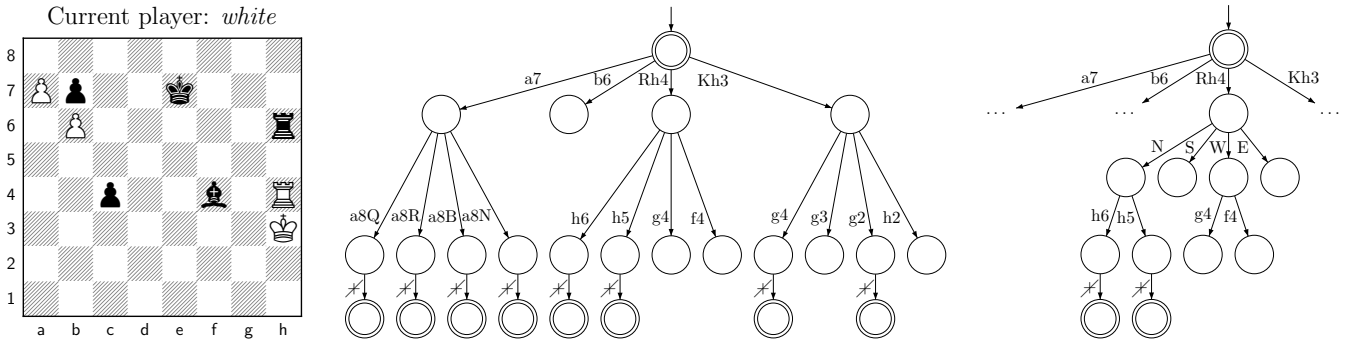


Figure 1: A Chess position (left) and the corresponding fragments of two semisplit games of a smaller (middle) and a larger (right) granularity. There are 8 legal moves in total (a7-a8Q, . . . , Kh3-g2 in the long algebraic notation), which form 8 edges in the standard game tree. Nodal states are marked with a double circle; $\not=$ indicates passed non-check king test. In the right semisplit game fragment, there are drawn 3 nodal, 9 intermediate, and 5 dead states.

plit MCTS uses both nodal and intermediate states as tree nodes. The expansion begins with the selection of an untried semimove (line 20). Dead states are never added to the MCTS tree. If the next state turns out to be dead, that semimove is removed from the list in the node, and the search goes back to the MCTS tree, so other untried semimoves are chosen (line 22). It can happen that all untried semimoves lead to dead states and the node becomes fully expanded; then the search continues according to the UCT policy.

Raw and nodal variants. There are two expansion variants, *raw* and *nodal*. The *raw* variant, given in Alg. 1, adds just one tree node as usual, either intermediate or nodal. However, this can leave semisplit MCTS behind orthodox MCTS in terms of expansion speed, as a single node in the latter corresponds to a path between nodal states in the former. In other words, when counting nodal states, orthodox MCTS expands faster. The *nodal* variant compensates this risk by adding the whole move (path) during a single expansion. This may slightly increase the quality of the search when these paths are long and, in particular, cannot be exploited due to slower expansion. However, the nodal variant may be slightly slower.

Final selection. There are several policies to select the final move to play. A common policy is to choose the one with the best average score (Winands 2017) among tried moves. When it comes to selecting the final move to play, semisplit MCTS greedily chooses the best semimove till the first next nodal state. If the move goes outside of the MCTS tree constructed so far, the raw variant chooses the remaining semimoves uniformly at random. This can happen when the branching factor is very large compared to the number of iterations. In the nodal variant this cannot occur, as all leaves added to the MCTS tree are nodal states.

Combined variants and roll-up. Possible variants of semisplit MCTS involve combining both designs and using them selectively in different phases. There are two natural variants: orthodox design in the MCTS tree phases (selection and expansion) combined with semisplit design in the simulation phase, and the opposite variant.

Another proposed variant is adaptive, applied in the MCTS tree phases. The idea is to use semisplit design at first, to improve efficiency and deal with potentially large branching factor, and then switch to orthodox design to provide more conservative evaluation. The *roll-up* variant uses semisplit design in the MCTS tree with the modification as follows. Whenever a node v of an intermediate state in the MCTS tree is fully expanded, i.e., all its children $v_1, \ldots, v_k$ were tried, the algorithm removes v and connects $v_1, \ldots, v_k$ directly to its parent v'. Then, the edges to them are submoves obtained from concatenating the submove from v' to v and the submoves from v to v_i. In this way, semisplit design is limited to the expansion phase, as the algorithm switches to orthodox design in the parts of the MCTS tree that become fully expanded. The roll-up variant can be used to test the impact of semisplit design in this phase alone.

Action-based heuristics. Common general knowledge-free enhancements of MCTS are online learning heuristics, which estimate the values of moves by gathering statistics. We test the two most fundamental methods, MAST and RAVE (Finnsson and Björnsson 2008; Gelly and Silver 2011; Winands 2017). MAST globally stores for every move (actually, for every set of moves with the same representation) the average result of all iterations containing this move; it is used in simulations in place of a uniformly random choice. RAVE stores similar statistics locally in each MCTS node for available moves and uses them to bias the choice in UCT. For both techniques, there are many policies proposed that differ in details.

MAST and RAVE can be adapted to semisplit in a straightforward way if semisplit design is used both in the MCTS tree and simulation. In combined variants, they require some adjustments, which rely on dividing (sub)moves into semimoves or merging semimoves into moves.

• *MAST-split* and *RAVE-split*: The basic modification of MAST adapted to semisplit simply stores separate statistics for each semimove. Then, when evaluating the score of a longer submove, we have to somehow combine the result from the scores of the included semimoves. Here, we propose the arithmetic mean of the scores of the semimoves. However, if a semimove has not been tried, then the maximum reward is returned as the final score of the whole submove. MAST-split can be applied to any variant of MCTS.

Table 1: Flat Monte-Carlo results (random simulations with gathering scores). The 2st and 3rd column show the speed measured in the number of computed respective nodal states and simulations per second. The 4th and 5th columns show the mean simulation depth measured resp. in nodal states and all states (computed dead states are also included). The last 6th column shows the branching factor calculated as the number computed (semi)moves divided by the number of states.

Game	Nodal states/sec.	Sim./sec.	Mean nodal states/sim.	Mean all states/sim.	Mean br. factor (all states)
Breakthrough (orthodox)	2,388,827 (100%)	37,272	64	64	25.69
Breakthrough (@Mod)	5,717,972 (239%)	78,120	73	157	7.71
Breakthrough (@ModShift)	7,558,711 (316%)	220,074	34	164	3.12
Chess (orthodox)	308,157 (100%)	976	316	316	22.83
Chess (@Mod)	1,556,036 (505%)	5,971	261	1,494	2.96
Chess (@ModPlus)	1,390,428 (451%)	5,341	260	2,194	2.31

RAVE-split works analogously, i.e., it splits every submove into single semimoves. Statistics for a semimove in a tree node are updated if this semimove was used explicitly at the node or later in the iteration, either directly or possibly as a part of a submove. RAVE-split is easily enabled only in the combinations using semisplit design in the selection and expansion phases, i.e. when the domain of semimoves in these phases correspond to the stored statistics.

Depending on the implementation, split variants can be much faster than regular heuristics due to a much smaller domain of semimoves (i.e., we can use faster data structures for storing statistics). However, obtained samples are less reliable, as semimoves carry less information than full moves.

• *MAST-join* and *RAVE-join*: These variants merge semimoves into full moves. MAST-join and RAVE-join store statistics only for whole moves. Of course, they are available only in the cases where we evaluate only moves in the corresponding MCTS phases.

• *MAST-context* and *RAVE-context*: To partially overcome the weakness of less reliable samples, we also propose *context* variants. They lead to storing sample values closer to those used in orthodox design. The main idea of MAST-context is entwined with *N-gram-Average Sampling Technique* (*NST*) (Tak, Winands, and Björnsson 2012), which gathers statistics for fixed-size sequences of moves. MAST-context maintains statistics for submoves of different lengths. When the iteration is over, for each move applied in the iteration, the statistics are updated not only for that move but also for each of its prefixes. The *context* of a state is the submove from the last preceding nodal state to this state. Thus, a context is a submove that is a prefix of some move. While selecting the best submove in the simulation phase, we use the statistics of the submoves concatenated to the current context. In RAVE-context, MCTS nodes store statistics for each child just as in the regular RAVE. The difference is in updating them; the statistic of a semimove is updated only if the iteration contains the same semimove played in the same context. The context variants are available for every variant of MCTS (in particular, for roll-up) because they store statistics for all submoves that may be ever needed.

• *MAST-mix* and *RAVE-mix*: For most MCTS variants, we have more than one choice for variants of action-based heuristics. This leads to the possibility of using more than one simultaneously. The variants of the heuristics differ in speed of gathering samples but also in their quality, e.g., MAST-split gathers samples faster than MAST-context, but they are less reliable. Hence, MAST-mix combines MAST-split with MAST-context, and RAVE-mix combines RAVE-split with RAVE-context. All statistics are updated separately according to both split and context strategies. For both heuristics, we have an additional parameter – the *mix-threshold*. During the evaluation, when the number (weight) of samples in the context heuristic is smaller than the mix-threshold, the score is evaluated according to the split variant; otherwise, the context statistics are used.

Implementation and Experiments

Our test set consists of 12 board games, well known in GGP (Amazons, Breakthrough, Breakthru, Chess, Chess without check, English Draughts, Fox And Hounds, Go, Knightthrough, Pentago, Skirmish, and The Mill Game).

Split Strategies. For a given game, there are many equivalent split versions. Usually, they are derived through a manual implementation of the algorithm computing legal moves and states. For the purposes of experiments, we use abstract game descriptions in the Regular Boardgames and derive semisplit games through *split strategies*, which are algorithms taking as the input a pure definition of the game rules (without any heuristic information for good playing). Thus, they can be considered knowledge-free and derive splittings automatically in a systematic way across different games.

Although split strategies might be interesting by themselves, they are developed here for the purposes of experiments. This part is to create an example environment for testing semisplit MCTS, and thus it is not the focus of the current paper. We note that using such split strategies would not be possible effectively without semisplit MCTS, as they introduce dead states. Similar effects could be obtained via other approaches; in particular, one could manually implement each semisplit game, apart from any GGP system.

Below, we describe the intuitive meaning of split strategies. Each split strategy is defined via a subset of three components.

• *Mod*: This is a basic component of relatively low granularity. Every semimove corresponds to an action modifying either a single square on the board or a variable; thus there is introduced a semimove for each elementary modification of the game state. A move in a chess-like game is split into two semimoves: one for selecting and grabbing a piece from the board and one for dropping it on the destination square. For modifying more squares, more semimoves are introduced

accordingly (e.g., Amazons). Also, it separates the final decision in a move, e.g., in Chess, the king check is performed in a final semimove; in Go, the player first chooses whether to pass or to put a stone. An example of the resulting semisplit game tree is shown in Fig. 1(middle).

• *Plus*: This component introduces a semimove for every single decision (branch) specified by the rules except those made iteratively. It commonly involves choosing the direction of the movement (Amazons, Chess, English Draughts) and move type (capture or two movements in Breakthru). It does not split decisions that are repeated, such as stopping on a square while moving in a chosen direction (e.g., rook upwards move). An example of the resulting semisplit game tree is shown in Fig. 1(right).

• *Shift*: This component splits square selection when it consists of more than one decision. It commonly separates the selection of a column and a row of the board (all games) and also divides movements of some pieces (e.g., knight – first long and then short hop).

By combining these components, we can build split strategies such as *Mod*, *ModShift*, or *ModPlusShift*. Additionally, the algorithm optimizes the semisplit game by removing some indecisive splits (i.e., semimoves that are always the only available choice) when possible.

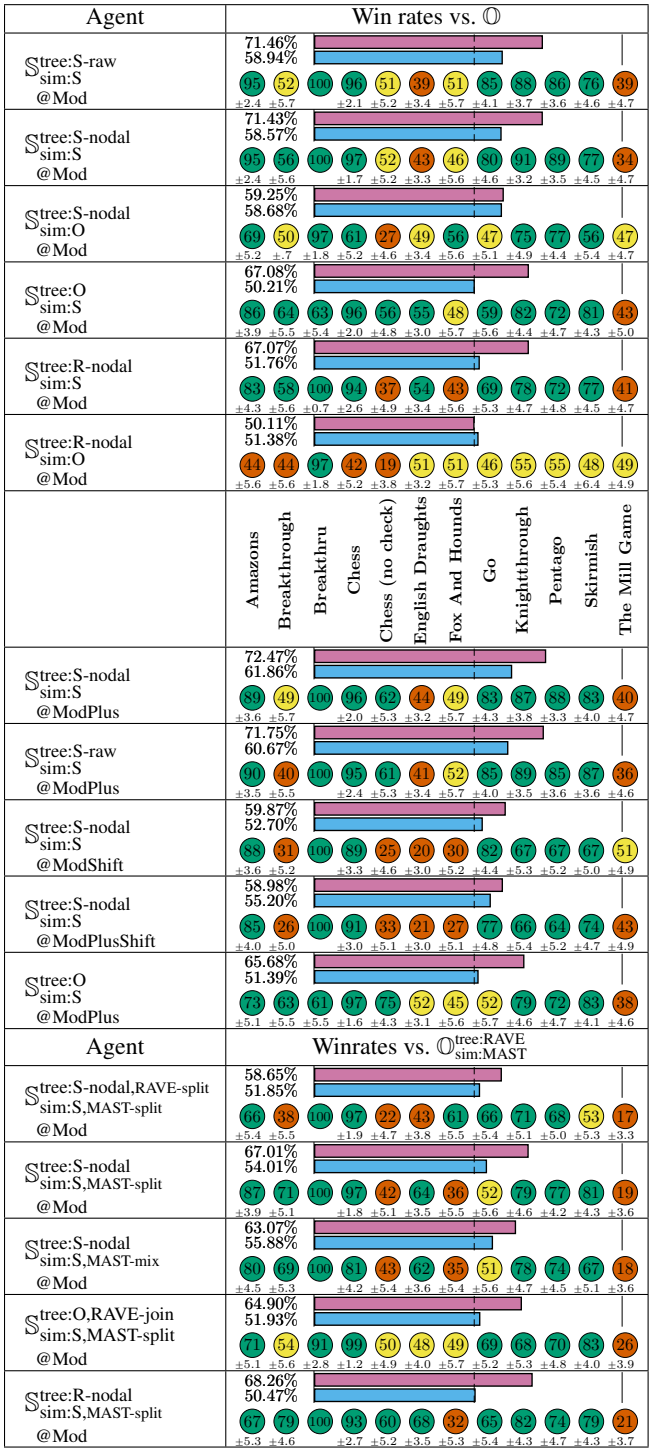
Results

Setup and notation. All parameters (e.g., the exploration factor, MAST and RAVE policies) were set according to the recommendations in the literature (Finnsson and Björnsson 2010; Sironi and Winands 2016) and keep the same for every agent. Semisplit agents were tested against the orthodox ones: both the vanilla MCTS agent (denoted by $\mathbb{O}$) and the enhanced MCTS agent ($\mathbb{O}^{\text{tree:RAVE}}_{\text{sim:MAST}}$) were used as baselines. Our semisplit agents are denoted by $\mathbb{S}$ with indices describing the variant: "S" means semisplit design and "O" means orthodox design, which can be independently used in the MCTS tree or in simulations; "R" is the roll-up variant. Raw and nodal variants and which variants of MAST and RAVE are used, if applied, are also indicated. Finally, the used split strategy is denoted after "@".

We tested agents in two settings: the *timed setting*, where agents have the same amount of time (0.5s) per turn, and the *fixed setting*, where agents have a limited number of nodal states to compute. These settings are correlated by letting the states limit equal to the number of states computed by the baseline orthodox agent performing within the given time limit. Hence, the fixed setting is used to observe the impact of semisplit design apart from efficiency benefits. Note that this is different than the common measurement with a fixed number of simulations and gives a better correspondence with the timed setting, as splitting alters simulation length.

Statistics. Tab. 1 shows illustratively how simulation statistics change under semisplit design. We observe a significant speed-up in terms of the number of traversed states, and thus of simulations in a given time limit. Obviously, semisplit design increases the depth of a simulation and reduces the mean branching factor. When the branching factor is already small, introducing more splits slows down. The real threshold strongly depends on the game and the implementation.

Table 2: Win rates of semisplit agents. The bars show the grand average in timed (upper, red) and fixed (lower, blue) settings. Timed results are also given separately for each game; green, yellow, and red indicate statistically significant *better*, *same*, and *worse* performance, respectively. A game is counted for better performance if the whole 95% confidence interval is above 50%; worse is symmetrical.



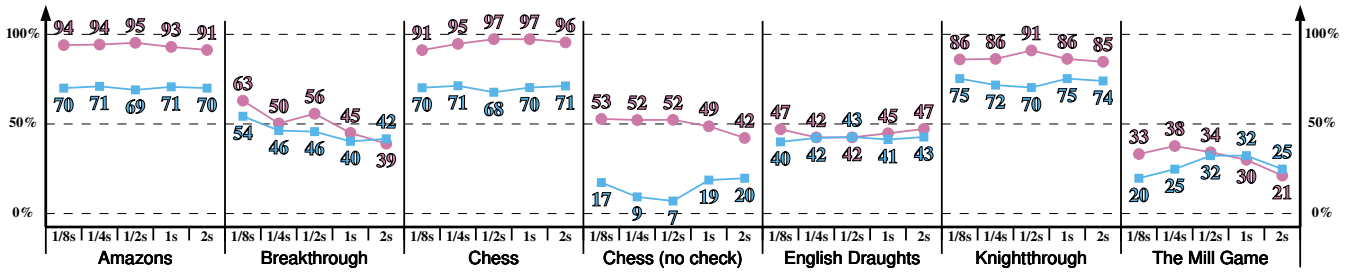


Figure 2: The results of $\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:S}}$@Mod versus $\mathbb{O}$ for different time limits (circle) and for equivalent states budgets (square).

Note that the mean depth of a simulation measured in nodal states is also altered, which can independently influence the number of simulations.

Comparison of agents. Table 2 contains three parts. The first part shows the results of variants of semisplit MCTS without action-based heuristics and using the Mod split strategy (one of the least granularity). All variants get better results than orthodox MCTS, except for $\mathbb{S}^{\text{tree:R-nodal}}_{\text{sim:O}}$@Mod. There is no visible difference between the raw and nodal variants. The efficiency benefits alone are most significant when the agent uses semisplit in simulations. We also see an advantage in the fixed setting when semisplit is applied in the MCTS tree. While $\mathbb{S}^{\text{tree:S-raw}}_{\text{sim:S}}$@Mod gets the largest mean, $\mathbb{S}^{\text{tree:O}}_{\text{sim:S}}$@Mod is the most robust, winning in ten games and losing only in one. The particular case of $\mathbb{S}^{\text{tree:R-nodal}}_{\text{sim:O}}$@Mod shows the effect of semisplit applied only to the expansion phase, which results in a kind of an unprunning method (Chaslot et al. 2008) – this is beneficial only for Breakthru, which has a very large branching factor.

The second part shows four representative combinations with split strategies of a larger granularity. Here, the nodal variant has a slight advantage over raw. The results follow the same pattern, yet ModPlus seems to be slightly better than Mod, but ModShift already worsens the results.

The third part shows selected combinations of agents equipped with RAVE and MAST against $\mathbb{O}^{\text{tree:RAVE}}_{\text{sim:MAST}}$. $\mathbb{S}^{\text{tree:S-nodal,RAVE-split}}_{\text{sim:S,MAST-split}}$@Mod is the simplest variant; it gives positive results but is not that strong as previously. Surprisingly, RAVE does not perform that well here; thus it is better to omit it. $\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:S,MAST-mix}}$@Mod gives the best results in the fixed setting, but the computation cost of MAST-mix does not overcome the benefits. The best and most robust agent among tested ones is the roll-up variant with semisplit simulations and MAST-split.

Fig. 2 shows how the win rate changes for different time limits and in relation to the fixed setting. In most cases, the results are kept consistent for different limits, especially in the games where semisplit is clearly beneficial.

Conclusions

We have introduced a family of Monte-Carlo Tree Search variants that work on semimoves – arbitrarily split game moves. The algorithm is based on the idea of lazy move computation, yet it has many possible variants. We applied split design for single-action games for the first time and revealed the impact on agents' results on a wider set of games (as previously, only Amazons was tested). Furthermore, we tested different configurations concerning the selective application in MCTS phases, split granularity, and variants of action-based heuristics. The developed framework allows testing the effects of (semi)splitting for more than 600 combined variants of semisplit, action-based heuristics, and split strategies, for any game described in RBG.

The impact of using semisplit is generally beneficial. First, it greatly improves search efficiency (typically, 3–5 times more nodal states/sec.). Moreover, for many games, the playing strength is improved over its orthodox counterpart, also when both algorithms are capped to the same performance. Even with general and blind split strategies, we were able to obtain win rates larger than 70% on about half of the test set (e.g., $\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:S}}$@Mod or $\mathbb{S}^{\text{tree:R-nodal}}_{\text{sim:S,MAST-split}}$@Mod). For comparison, the benefits are larger than those from adding action-based heuristics to the vanilla orthodox agent (the mean win rate of $\mathbb{O}^{\text{tree:RAVE}}_{\text{sim:MAST}}$ vs. $\mathbb{O}$ on our test set is 53% in timed and 65% in fixed setting).

More detailed results show that using semisplit in simulations gives a large efficiency boost and generally is not harmful in terms of their quality. This may be useful also apart from game playing, as simulations are applied to many single-player problems (Wang et al. 2020). On the other hand, using semisplit in the MCTS tree gives some benefits in the quality of iterations, yet it is riskier, as sometimes it is consistently harmful (e.g., The Mill Game). However, on average, almost none of the tested variants is worse in the fixed setting than the orthodox baseline.

The conducted research can be seen as pioneering, as there are many directions for future research, e.g.:

- There should be developed methods for choosing the most suitable semisplit variant and split strategy for a given game.
- The used parameters were the same for both agent types and were tuned rather for orthodox agents according to the literature. This indicates that after a tuning, semisplit should achieve even better results.
- We focused on the practical aspect, yet there are interesting theoretical questions as to how strongly splitting the game can distort the agent's results, and how hard is the problem? to determine whether splitting will be beneficial.
- Semisplit can be combined with prior knowledge (Gelly and Silver 2007) or neural networks. It can reduce action space and improve efficiency, especially when simulations are combined with neural networks (Cotarelo et al. 2021).

Acknowledgements

This work was supported in part by the National Science Centre, Poland under project number 2021/41/B/ST6/03691 (Jakub Kowalski, Marek Szykuła).

References

Baier, H.; and Winands, M. H. M. 2018. MCTS-minimax hybrids with state evaluations. *JAIR*, 62: 193–231.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE TCIAIG*, 4(1): 1–43.

Cazenave, T. 2015. Generalized rapid action value estimation. In *IJCAI*, 754–760.

Chaslot, G.; Winands, M. H. M.; van den Herik, H. J.; Uiterwijk, J.; and Bouzy, B. 2008. Progressive strategies for Monte-Carlo tree search. *New Mathematics and Natural Computation*, 4(3): 343–357.

Childs, B. E.; Brodeur, J. H.; and Kocsis, L. 2008. Transpositions and move groups in Monte Carlo tree search. In *IEEE Symposium On Computational Intelligence and Games*, 389–395.

Cotarelo, A.; García-Díaz, V.; Núñez-Valdez, E. R.; González García, C.; Gómez, A.; and Chun-Wei Lin, J. 2021. Improving Monte Carlo Tree Search with Artificial Neural Networks without Heuristics. *Applied Sciences*, 11(5).

Finnsson, H.; and Björnsson, Y. 2008. Simulation-Based Approach to General Game Playing. In *AAAI*, volume 8, 259–264.

Finnsson, H.; and Björnsson, Y. 2010. Learning Simulation Control in General Game Playing Agents. In *AAAI*, 954–959.

Fotland, D. 2006. Building a World-Champion Arimaa Program. In *Computers and Games*, volume 3846 of *LNCS*, 175–186.

Geißer, F.; Speck, D.; and Keller, T. 2020. Trial-based Heuristic Tree Search for MDPs with Factored Action Spaces. In *Thirteenth Annual Symposium on Combinatorial Search*.

Gelly, S.; and Silver, D. 2007. Combining online and offline knowledge in UCT. In *Proceedings of the 24th International Conference on Machine Learning*, 273–280.

Gelly, S.; and Silver, D. 2011. Monte-Carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence*, 175(11): 1856–1875.

Genesereth, M.; Love, N.; and Pell, B. 2005. General Game Playing: Overview of the AAAI Competition. *AI Magazine*, 26: 62–72.

Goldwaser, A.; and Thielscher, M. 2020. Deep Reinforcement Learning for General Game Playing. In *AAAI*.

Hufschmitt, A.; Vittaut, J.-N.; and Jouandeau, N. 2019. Exploiting Game Decompositions in Monte Carlo Tree Search. In *Advances in Computer Games*, 106–118. Springer.

Justesen, N.; Mahlmann, T.; Risi, S.; and Togelius, J. 2017. Playing Multiaction Adversarial Games: Online Evolutionary Planning Versus Tree Search. *IEEE Transactions on Games*, 10(3): 281–291.

Kaiser, Ł.; and Stafiniak, Ł. 2011. First-Order Logic with Counting for General Game Playing. In *AAAI*, 791–796.

Kloetzer, J.; Iida, H.; and Bouzy, B. 2007. The Monte-Carlo approach in Amazons. In *Proceedings of the Computer Games Workshop*, 185–192.

Kocsis, L.; and Szepesvári, C. 2006. Bandit Based Monte-carlo Planning. In *European Conference on Machine Learning*, 282–293.

Kowalski, J.; Mika, M.; Pawlik, W.; Sutowicz, J.; and Szykuła, M. 2021a. Regular Boardgames – source code (version 1.3). https://github.com/marekesz/rbg.

Kowalski, J.; Mika, M.; Pawlik, W.; Sutowicz, J.; Szykuła, M.; and M., W. M. H. 2021b. Split Moves for Monte-Carlo Tree Search. ArXiv:2112.07761 [cs.AI].

Kowalski, J.; Mika, M.; Sutowicz, J.; and Szykuła, M. 2019. Regular Boardgames. In *AAAI*, 1699–1706. https://arxiv.org/abs/1706.02462.

Love, N.; Hinrichs, T.; Haley, D.; Schkufza, E.; and Genesereth, M. 2006. General Game Playing: Game Description Language Specification. Technical report, Stanford Logic Group.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533.

Ontañón, S. 2017. Combinatorial Multi-armed Bandits for Real-Time Strategy Games. *J. Artif. Intell. Res.*, 58: 665–702.

Perez, D.; Samothrakis, S.; Lucas, S.; and Rohlfshagen, P. 2013. Rolling horizon evolution versus tree search for navigation in single-player real-time games. In *GECCO*, 351–358.

Perez, D.; Samothrakis, S.; Togelius, J.; Schaul, T.; and Lucas, S. M. 2016. General Video Game AI: Competition, Challenges and Opportunities. In *AAAI*, 4335–4337.

Piette, E.; Soemers, D. J.; Stephenson, M.; Sironi, C. F.; Winands, M. H. M.; and Browne, C. 2020. Ludii - The ludemic general game system. In *ECAI 2020*, 411 – 418.

Rasmusen, E. 2007. *Games and Information: An Introduction to Game Theory*. Blackwell, 4th ed.

Roelofs, G.-J. 2017. Pitfalls and Solutions When Using Monte Carlo Tree Search for Strategy and Tactical Games. *Game AI Pro 3: Collected Wisdom of Game AI Professionals*, 343—-354.

Saito, J.-T.; Winands, M. H. M.; Uiterwijk, J. W.; and van den Herik, H. J. 2007. Grouping nodes for Monte-Carlo tree search. In *Computer Games Workshop*, 276–283.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; et al. 2018. A general reinforcement learning algorithm

that masters chess, shogi, and Go through self-play. *Science*, 362(6419): 1140–1144.

Sironi, C. F.; and Winands, M. H. M. 2016. Comparison of rapid action value estimation variants for general game playing. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8.

Świechowski, M.; Godlewski, K.; Sawicki, B.; and Mańdziuk, J. 2021. Monte Carlo Tree Search: A Review of Recent Modifications and Applications. *arXiv preprint arXiv:2103.04931*.

Tak, M. J.; Winands, M. H. M.; and Björnsson, Y. 2012. N-grams and the last-good-reply policy applied in general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2): 73–83.

Van Eyck, G.; and Müller, M. 2011. Revisiting move groups in monte-carlo tree search. In *Advances in Computer Games*, 13–23.

Wang, H.; Preuss, M.; Emmerich, M.; and Plaat, A. 2020. Tackling Morpion Solitaire with AlphaZero-like Ranked Reward Reinforcement Learning. In *2020 22nd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 149–152. IEEE.

Winands, M. H. M. 2017. Monte-Carlo Tree Search in Board Games. In *Handbook of Digital Games and Entertainment Technologies*, 47–76.