Monte Carlo Tree Search With Reversibility Compression

Michael Cook
School of Electronic Engineering and Computer Science
Queen Mary University of London
mike@possibilityspace.org

***Abstract*—Monte Carlo Tree Search (MCTS) has been shown to be an effective algorithm for solving search problems in the absence of heuristics. MCTS performs best in spaces where every action has a significant impact on the outcome of the search. In many domains, however, particularly single-player games, many actions have little impact on the outcome, which makes MCTS perform poorly without heuristic support. To address this deficiency in such sparse-impact search problems, we introduce *MCTS with Reversibility Compression*, or MCTS-R, which uses the notion of *action reversibility* to compress MCTS trees as they are constructed, without loss of information. This not only reduces the memory footprint of the search tree, but also accelerates search by preventing the duplication of already-explored states, and increasing the attention paid to significant actions. We show that our approach outperforms several comparable algorithms for solving sparse-impact search problems.**

I. Introduction

Developing game-playing AI agents is vital for a wide range of game AI research, including procedural content generation, automated game design and player modelling. It is also important for game development too, as a tool for testing and to provide characters to challenge, co-operate and entertain the player. Many approaches to creating AI agents for games leverage expert knowledge about the game being played – for example, an agent that plays Starcraft 2 might be provided with pre-set build orders. Expert knowledge often focuses on simplifying the search space. A navigation mesh is an example of such expert knowledge – rather than ask an AI character to navigate around an unseen environment, we provide a simplified map of the space so it can plan routes around the space and play the game more effectively.

There are many situations where we might want an AI agent to play a game it has never seen before. For example, a co-creative tool that must play and evaluate a game design which is constantly being changed by a human user; or in automated game design, where an AI system must generate and test games that have never been seen before. In such situations we rely on *general game-playing* techniques. However, true general game-playing is a serious challenge, and the most effective approaches are often computationally expensive.

A commonly-used technique for general game-playing is Monte Carlo Tree Search, or MCTS, a type of tree search algorithm. We define the problem of tree search as the search for a path of nodes through a tree $\mathcal{T} = (V, e)$ beginning at the root vertex of the graph and terminating at a leaf vertex. Each leaf vertex $l \in V$ is assigned a value $s \in \mathbb{R}$, its *score*, describing the quality of the path or the leaf node. Common aims for search include finding a path to the highest-valued leaf, or finding the shortest path to a maximally-valued leaf. An example of the former might be achieving a high-score in an arcade game, while an example of the latter might be finding the shortest path to a winning position in Chess.

MCTS performs well even with little or no knowledge about the game it is playing [4]. However, due to the nature of tree search algorithms such as MCTS, it suffers in games where there are many actions which do not advance the state of the game towards success or failure. Physical games, such as Chess, Go or Poker, tend to have very few such actions. Most actions made by a player advance the game in some way. Videogames, however, are rife with such actions.

In [8] we describe such actions as *reversible* actions. An action a in a game state s is reversible if there exists a sequence of one or more actions which, when taken after a, return the simulation to a state equal to s under some equivalence relation $\equiv$. In this paper we consider a class of tree search problems with a high ratio of reversible to non-reversible actions in their search space – we call such problems *sparse-impact* problems. Examples of problems in this class include automating the playing of many single-player games, including modern hit games such as *Monument Valley*, and classic Atari games such as *Adventure*, *Montezuma's Revenge* and *Miniature Golf*, as well as single-agent navigation tasks. We show that standard MCTS performs poorly on such problems, and propose a novel variant of MCTS, which uses action reversibility in order to dynamically restructure the search tree as it is built. We motivate and describe our algorithm, MCTS-R, and evaluate it on a standard general game-playing test set, comparing it against other MCTS approaches. We show that it outperforms both in time ranking and search iteration ranking, and that it achieves high tree compression without information loss.

II. Background

A. Reversibility Compression

In [8] we describe *action reversibility* as a metric for compressing state space graphs (SSG) for games. An SSG, $\mathcal{G} = (S, t)$ is a directed graph in which the vertices, S,

The author was supported by the Royal Academy of Engineering's Research Fellowship scheme.

978-1-6654-3886-5/21/$31.00 ©2021 IEEE

represent game states and the edges, t, represent actions that transition the game between two states. Action transitions are defined as a labelled relation on game states: given states $s, s' \in S$, $(s, a, s') \in t$ is a transition stating that taking the action a in state s transitions the game to the state s'. Let $\mathcal{A}$ be the set of all actions.

An action is *reversible* if all of its effects can be undone by a sequence of one or more actions. More formally, there are two types of reversibility. A transition $(s_1, a, s_2) \in t$ is *immediately reversible* if $\exists a'.(s_2, a', s_3) \in t$ such that $s_3 \equiv s_1$, where $\equiv$ is an equivalence relation on game states. They use a strong equivalence relation, where $\forall s, s' \in S, s \equiv s'$ iff there is *no measurable difference* between the data describing s and s'. They further suggest that weaker equivalence relations could be used for different applications of reversibility.

Some actions are not immediately reversible but can be reversed by taking multiple actions. A transition $(s_1, a_1, s_2) \in t$ is said to be *eventually reversible* iff:

$$\exists \{a_2, ..., a_n\} \subset \mathcal{A}, \{s_2, ..., s_{n+1}\} \subset S. \\ \{(s_2, a_2, s_3), ..., (s_n, a_n, s_{n+1})\} \subset t \wedge s_{n+1} \equiv s_1 \quad \text{(D1)}$$

The authors subsequently introduce the notion of a *hyperstate space graph* (HSG). An HSG is constructed from an SSG by merging together states in the SSG that are connected together by reversible actions, a process we describe below. The resulting states are called *hyperstates*. Given an SSG, $\mathcal{G}{=}(S, t)$, its associated HSG is given by $\mathcal{H}{=}(H, c)$, where H is a set of *hyperstates* and c is a labelled relation describing action transitions on hyperstates, such that:

1) Each hyperstate $h \in H$ contains a non-empty *set* of states in G.
2) Each state $g \in G$ appears in *exactly one* hyperstate.
3) Each hyperstate contains states that are mutually reachable via reversible actions. Formally, $\forall h \in H$ and $\forall g, g' \in G$, if $g, g' \in h$, then there is a sequence of reversible actions from g to g', and from g' to g.
4) The HSG action transitions are lifted from SSG actions i.e., if $(h, l, h') \in c$, then there exists $g \in h$ and $g' \in h'$ such that $(g, l, g') \in t$, and vice versa.

For a fuller definition of HSG construction, we refer the reader to [8]. The intuition behind reversibility and merging states to construct HSGs is to reduce the significance of reversible actions, which can easily be undone and thus are less relevant than non-reversible actions, which must be committed to permanently in any plan or path. The authors propose this as a technique for automating game analysis, and suggest applications to general game-playing as a point of future work [11]. We build on this work here, showing that this graph transformation can be applied to tree search and improve the performance of search algorithms such as MCTS.

B. Problem Definition

We are concerned with a subset of tree search problems which we call *sparse-impact* search problems. By *sparse-impact* we mean that there is a non-empty subset of actions in the search space that are all reversible. The larger this subset is as a proportion of the set of actions, the more sparse the problem is considered to be.

Fig. 1. An immediately reversible action in Sokoban.

As mentioned previously, a common algorithm employed for tree search problems is Monte Carlo Tree Search (MCTS). This has been shown to be effective at playing two-player adversarial games both with and without game-specific heuristics to guide play [4], and formed part of the AlphaGo system which achieved superhuman play at Go [23]. MCTS has also been used to play single-player games, including those with stochasticity [19] and those without [21].

The standard MCTS algorithm has four stages: *selection*, *expansion*, *simulation* and *backpropagation*. Beginning at the root of the tree, the current search tree is descended using a *selection* strategy until a leaf node is reached. This leaf node is then *expanded* by generating its descendant nodes (states reachable from this leaf node state by a single action). One such descendant node is then selected for *simulation*, which chooses actions according to a simulation strategy until a terminal state is reached or a stopping condition is met. Finally, the result of the simulation is *backpropagated* up the tree, providing additional reward information which informs the next round of node selection.

Some of the best-known applications of MCTS have been to traditional boardgames such as Go and Chess. A commonly-shared property of these games is that moves are usually irreversible, and in the case where reversible moves are possible, they are unlikely to be encountered in optimal play. In Go, for example, special *Ko* rules prevent players from repeating board states indefinitely. In Chess, reversible moves are considered sufficiently abnormal that a game ends prematurely in a draw if the same game state is encountered three times. Thus, in any state, most or all actions irreversibly progress the game towards a conclusion. From the perspective of MCTS, this means that the expansion step, in which new nodes are added to the tree, will often be adding game states which have not occurred yet in that branch of the search tree. Repeated states can be encountered in other branches of the tree, where the same state is reached through a different sequence of actions. For tree search algorithms, this is often solved through techniques such as transposition tables, as in [13], and validating the correctness of paths that include repeated states, as in [17].

In sparse-impact search problems there are many actions

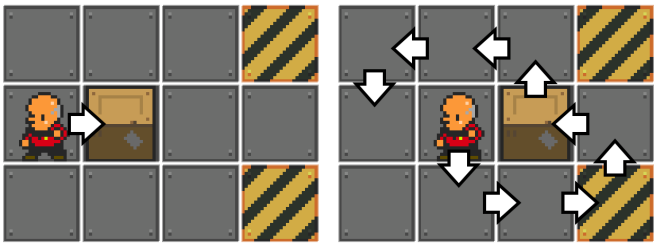

Fig. 2. An eventually reversible action in Sokoban

which do not necessarily progress the system towards a terminal state. By definition, if a *reversible* action a taken from a state s transitions the system to a state s', then at least one action available in state s' must be part of a reversible action chain that leads to a state s'' such that $s \equiv s''$. In a search tree, such as those constructed by MCTS, reversible action chains can repeat indefinitely, causing the search process to revisit duplicates of states that already occur earlier in the tree. Search effort expended on expanding nodes with reversible actions is less productive, therefore, as it leads to areas of the search space that have already been explored.

Solving levels of the videogame Sokoban is an example of a sparse-impact problem, for which determining the solvability has been shown to be NP-hard [10]. In Sokoban, the player controls a worker in a warehouse who must push crates into certain configurations, usually in tight spaces that force the player to consider the ordering of pushes. Moving the player in Sokoban into an empty tile is an *immediately reversible* action, as shown in fig. 1. Moving a crate may be eventually reversible, as shown in fig. 2. It may also be an irreversible move, e.g. pushing a crate into a corner of a room is irreversible, because the player cannot get behind the crate to push it back. Specialised MCTS agents have been developed for Sokoban [18]. However, instead of solving the level from the perspective of the player's movements, they instead solve an action-abstracted representation only considering the movement of crates. This greatly simplifies the problem, and in addition, the invention of this heuristic requires expert human knowledge of the game in advance of designing the solver. For many problem domains, such as general game-playing [11] or automated game design [7], no advance expert knowledge about the problem is available.

III. MCTS-R

In this section we describe *MCTS with Reversibility Compression* or MCTS-R. This applies ideas from reversibility reduction to compress the MCTS search tree as it is constructed. In doing so, we gain two advantages: first, we prevent the repeated expansion of states that do not meaningfully differ from previously-visited states; and second, we elide less important actions so the search tree is focused on more significant (i.e. non-reversible) decisions. This accelerates the search process over sparse-impact spaces, i.e., for problems with a large proportion of reversible actions.

Algorithm 1 Pseudocode for reversibility checking.

```
function CHECKCOMPRESSION(node)
    topOfChain ← node.parent
    actions ← [ ]
    while topOfChain ≠ nil do
        if node.state in topOfChain.hashes then
            break
        end if
        APPEND(actions, topOfChain.actions)
        topOfChain ← topOfChain.parent
    end while
    if topOfChain = nil then
        return false
    end if
    COMPRESS(node, topOfChain, actions)
    return true
end function
```

A. Tree Node Modifications in MCTS-R

In the basic MCTS algorithm, each node in the search tree is typically represented by a simple data structure storing the action taken to reach this node from its parent, as well as links to child, sibling and parent nodes. Thus, starting at the root of the tree and with the system in its initial state, each node's action can be applied to the state as the algorithm walks the tree, in order to obtain the system state corresponding to any leaf node in the search tree. An alternative approach is to have each node record a copy of the game's state, which trades off memory usage for performance.

In MCTS-R we extend the tree node data structure in two ways. First, rather than storing a single action, the node stores a *list* of actions. Initially this list contains a single action, as in a standard MCTS system. As the tree is compressed during search, by merging together nodes connected through reversible actions, some nodes will have additional actions added to their list. These actions are added to maintain an invariant relationship between a node, s, and its parent, s_p, namely that applying the list of actions in s, in sequence, to the system state represented by the node s_p, will yield the system state represented by the node s.

The second modification to the tree node is a list of system state hashes. This list initially has one element: a hash of the state represented by the node. Under certain conditions, some nodes are removed from the search tree and merged into other nodes. These removed nodes add their hashes to the hash list of the node they merge into. These hashes are used to check for further compression opportunities later in the search. We explain the function of both the action list and the state hash list in the remainder of this section.

B. Reversibility Checking

MCTS-R modifies the MCTS algorithm during the Selection phase, in which the algorithm traverses the tree, selecting nodes based on a tradeoff of their average score and the number of times they have been selected previously. Each

Algorithm 2 Pseudocode for reversibility compression.

```
function COMPRESS(node, tgt, ac)
    DETACH(node)
    for child in node.children do
        if node.state = tgt.state then
            COMPRESS(child, tgt, [ ])
        else if node.state in tgt.hashes then
            APPEND(ac, node.actions)
            COMPRESS(child, tgt, ac)
        end if
        modified ← False
        for tChild in tgt.children do
            if child.state in tChild.hashes then
                if node.actions < tChild.actions then
                    tChild.actions ← node.actions
                end if
                COMPRESS(child, tChild, [ ])
                modified ← True
            end if
        end for
        if not modified then
            for tChild in tgt.children do
                if child.state in tChild.hashes then
                    APPEND(ac, node.actions)
                    COMPRESS(child, tgt, ac)
                    modified ← True
                end if
            end for
        end if
        if not modified then
            DETACH(child)
            ATTACH(child, target)
            UPDATENODE(child, node, ac)
        end if
    end for
end function
```

time a node is selected, MCTS-R performs a *reversibility compression check* on the selected node. Algorithm 1 shows an outline of this process. Formally, for a node s, let $h(s)$ be the hash of the state represented by s, and let $\mathcal{H}(s)$ be the list of hashes contained within s as described above. Let the antecedent of a node s_n be denoted s_{n-1}. In order to check whether a node s_n can be compressed, we search every antecedent node of s_n to see if its hash is contained within any of the state hash lists maintained by the antecedent nodes:

$$\exists i.0 \leq i < n \wedge h(s_n) \in \mathcal{H}(s_i)$$

The compression check aims to establish whether any of the node's antecedents contain a state hash that is the same as the selected node's state hash. Beginning at the parent of the node being checked, we iteratively check the selected node's hash against the list of hashes in each antecedent node. If no matches are found, then the compression check fails. However, if some antecedent node is found to contain the same hash as the node being checked, then we can compress the tree.

To more formally describe this check, and to contextualise it in the terms of reversibility compression, let $\mathcal{T} = (S, t)$ be a search tree as defined above.

Let $s_n \in S$ be a node in a search tree which is being checked for compression, and let $s_i \in S$ be some antecedent node in $\mathcal{T}$ where $i < n$. Thus, there is some sequence of actions that transforms s_i into s_n, which is equivalent to walking the tree. More formally:

$$\exists\{a_i, ..., a_{n-1}\} \subset A, \{s_i, ..., s_n\} \subset S. \\ \{(s_i, a_i, s_{i+1}), ..., (s_{n-1}, a_{n-1}, s_n)\} \subset t$$

Suppose that s_n and s_i are shown to be equivalent under some relation $\equiv$. Then we have $(s_{n-1}, a_{n-1}, s_i) \subset t$. With this new piece of information, we can restate the above as follows:

$$\exists\{a_i, ...a_{n-1}\} \subset A, \{s_i, ...s_n\} \subset S. \\ \{(s_i, a_i, s_{i+1}), ..., (s_{n-1}, a_{n-1}, s_n)\} \subset t \wedge\ s_i \equiv s_n$$

We can see here the structural resemblance between this statement about a search tree, and the definition of eventual reversibility in Definition D1. Thus, if we establish that a selected node is equivalent to some antecedent node in the search tree, we have shown that it is part of a chain of eventually reversible actions.

C. Reversibility Compression

When a cycle of reversible actions is detected at node s_n in the tree, we attempt to compress the cycle into the node nearest the root of the tree, s_i. Given a search tree $\mathcal{T} = (S, t)$ as defined earlier, let $\mathcal{T}[s]$ denote the subtree of $\mathcal{T}$ rooted at $s \in S$. Reversibility compression transforms s_i into a new node s_i' with the following postconditions holding:

- $\mathcal{T}[s_i'] = \mathcal{T}[s_i] \setminus \{s_{i+1}, ..., s_n\}$ i.e., search tree nodes $\{s_{i+1}, ..., s_n\}$ are no longer in the tree.
- The node s_i' contains the hashes of the states $\{s_{i+1}, ..., s_n\}$ in its hash list.
- Define D, the set of *dangling* nodes, as the set of descendant nodes of s_{i+1} that are *not* in the reversibility chain, i.e. $\mathcal{T}[s_{i+1}] \setminus \{s_{i+1}, ..., s_n\}$. Each node $s_d \in D$ is inserted into the new subtree $\mathcal{T}[s_i']$ such that its relationship to s_i' mirrors its previous relationship to its nearest antecedent in $\{s_1, ..., s_n\}$.

To more formally expand this last point, define the set of nodes in the reversibility chain as $S_c = \{s_{i+1}, ..., s_n\}$. For any $c \in S_c$, define the *detached subtree* $\mathcal{T}_c^d$ as $\mathcal{T}[c] \setminus S_c$. When a given descendant of this chain node, $s_c^d \in \mathcal{T}_c^d$, is reinserted into the search tree at $\mathcal{T}[s_i']$, the depth of s_c^d relative to s_i' is the same as its previous depth relative to c. To ensure this, MCTS-R steps through the detached subtree and check each node in turn, reinserting it into the tree if no node already exists with the same state hash, or skipping it if an existing node is found at the same depth in $\mathcal{T}[s_i']$.

Algorithm 2 shows a pseudocode outline of the compression process. Beginning at the last node in the reversibility chain – s_n in our running example – we iteratively detach the subtree

rooted at this node, $\mathcal{T}[s_n]$, from the larger subtree rooted at s_i. We add the hash of s_n to the list of hashes in s_i, since this node now represents all of the compressed states. MCTS-R then walks the detached subtree in a breadth-first fashion, attempting to reinsert each node into the search tree, rooted at s_i. This process may recurse in order to insert descendants of s_n into the search tree relative to descendants of s_i.

When reinserting a node, s_j, into the subtree rooted at some descendant of s_i, s_n^i, if we find that there is no existing node at this depth with the same hash as s_j, we can insert s_j and all of its descendants as a whole new subtree that is a direct descendant of s_n^i, as it represents paths which are not represented here. Alternatively, if a node with the same hash exists already as a descendant of s_n^i, or a node contains the inserted node's hash in its hash list, then we recurse the insertion process and attempt to reinsert s_j's descendants into s_n^i or one of its descendants, according to conditions outlined in Algorithm 2.

By performing this compression, we are exploiting the fact that any of the states $\{s_{i+1}, ..., s_n\}$ are reachable from one another without permanently altering the game's state. Thus, by reaching the first state in the cycle, s_i, any of these other states are trivially accessible. The search process should only be concerned with actions which are not currently known to be reversible, namely the nodes in $\{s_{i+1}, ..., s_n\} \setminus \mathcal{T}[s_{i+1}]$. Hence, we insert them in the tree as relative descendants of s_i instead, and allow the algorithm to treat them as if they are immediately accessible from s_i.

This transformation of the tree removes many nodes from the search tree that are irrelevant, and thus gives greater prominence to the remaining nodes which represent areas of the search space that are either more significant to the outcome of the search, or less well explored (since they may prove to be reversible with more search effort). However, by removing and transposing nodes in the search tree, we are losing crucial information about the structure of the search space, which makes it hard to reconstruct paths from root to leaf after the algorithm is complete. This necessitates a maintenance step to update information in the tree, which we describe below.

D. Maintaining Action Lists

Recall that in an ordinary MCTS search tree, a node s in a tree contains an action a such that taking that action in its antecedent state, s', transitions the system to the state described by s. In other words, given a tree $\mathcal{T} = (S, t)$, $(s', a, s) \in t$. However, restructuring a tree in the way described above would break this invariant for any nodes which are transposed as part of the compression process.

In order to maintain this relationship between nodes and their parents, we earlier stated that we extended the description of a search tree node so that it maintains a *list* of actions rather than a single action. When we restructure the tree, we update this list for affected nodes so that it always reflects the shortest known path to reach the current node from its new immediate parent. To perform this update, we calculate an updated action list during the compression phase.

Let us return to our running example of a search tree $\mathcal{T} = (S, t)$, containing a set of nodes in a reversibility chain, $\{s_i, ..., s_n\}$, where s_i is the compression target, and the remaining nodes, $S_c = \{s_{i+1}, ..., s_n\}$, are to be compressed. For any $s_j \in S_c$, the forest (a collection of trees) $\mathcal{T}[s_j] \setminus S_c$ will be reinserted into the search tree with s_i as the new parent. Each child of s_j will prepend to its action list a new list of actions, C, defined as follows:

$$C = \{a \mid \forall p, q.i \leq p, q \leq j \wedge (s_p, a, s_q) \in t\}$$

That is, C is the list of actions required to walk the *original* search tree from s_i to s_j. This new action chain is only prepended if the node is reinserted fully into the search tree underneath s_i. If a child of s_i already exists representing the same system state, we compare the existing child's action chain with the new node's action chain, and only replace the action chain if the new one is shorter, since this describes a shorter sequence of actions that achieves the same outcome. With this final modification, the tree is now restructured in such a way that search can resume. MCTS-R has removed all nodes that were superfluous to the search process; and has updated action chains and hash lists to reflect the alterations made to the structure of the tree.

IV. Related Work

A. Abstraction and Refinement

Abstraction and refinement is an optimisation technique that accelerates searches on graphs by constructing an abstracted version of the base graph, performing search on the abstracted graph form, and then *refining* the abstract solution into a full solution that can be applied to the base, original graph [15].

Graph abstraction and refinement has been applied to games, particularly to the problem of *pathfinding*. In [25] Sturtevant et al present an abstraction technique in which a game space is overlaid with a grid, effectively abstracting the continuous space to a simpler representation. A given point within the game world is mapped to the nearest point on the grid. This approach has been extended to suit many different pathfinding scenarios [5], [14], including abstracting directed graphs and incorporating limited actor state representation, as in [12].

The process of compressing an MCTS tree as it is constructed can be thought of as a process of abstraction. However, unlike classical approaches to abstraction and refinement, the compressed MCTS tree is an end in itself – it does not require refinement to be useful, and is designed to be used, analysed and studied in its compressed form. We are primarily interested in the process of building the abstracted graph and the nature of the abstracted graph, rather than the utility of the abstraction in querying the original graph. Some results from abstraction and refinement literature may be applicable to improve the MCTS-R compression process in the future, such as [2] which parallelises the decomposition of strongly connected components. Such an approach could make tree compression more efficient, but it would need to be studied in the context of trading off exploitation (to avoid compressing parts of the tree which are not being explored anyway).

B. Sparse-Impact Game Playing

In [18] the authors describe two systems for solving Sokoban levels, using MCTS and IDA*, a variant of A* search. The authors augment their MCTS approach with optimisations including *cycle detection*, whereby a table of all searched states is maintained, and if a new child node would cause a cycle in the current branch, it is not generated.

Cycle detection is similar to the identification of reversible actions in MCTS-R. However, our approach differs significantly because of the compression step which follows the detection of a reversible action. By restructuring the search tree, we dramatically reduce the number of nodes in the tree; elevate more significant actions so they are closer to the root; and make the detection of reversible actions more likely in other branches by maintaining hash lists of detected cycles.

Transposition tables are a commonly-employed technique in AI for game-playing, where game states are recorded and information about them reused when they are encountered subsequently in the search process [13]. Transposition tables have been applied to MCTS, such as in [6] and [16] where it is used for playing Go. In these applications, backpropagation updates a transposition table, rather than a node in the search tree, so the results of a simulation are reused anywhere in the search tree this state occurs. Our approach differs by focusing on reversible chains within a subtree, rather than single-state repetition across the tree breadth, and by restructuring the tree itself rather than abstracting collected information.

State and action abstraction are common techniques for improving the performance of tree search algorithms on large-scale problem domains, by treating clusters of states and actions as single units [9]. [1] show that MCTS can be adapted to search through abstracted domains and even outperform MCTS applied to the original non-abstracted problem. These approaches are powerful, but rely on pre-existing abstractions of the problem at hand.

V. Evaluation

MCTS-R is designed as a general-purpose algorithm for sparse-impact search problem. Common test domains for standard MCTS include Chess and Go, however for the purposes of deterministic, single-agent search, a classic problem is the puzzle game Sokoban, which we use here as a sparse-impact problem domain. *Sokoban* is a single-player puzzle game in which the player controls a warehouse worker who must push a series of crates onto goal locations. A crate can only be pushed, not pulled, and multiple crates cannot be pushed in a row. A puzzle is solved when *every* goal location is covered by a crate. Sokoban is NP-Hard [10], and is a common baseline game used to test general game-playing agents [11]. We used the Microban collection of 155 Sokoban levels. This test set is used both in [18], and as a source of Sokoban levels in the General Video Game AI Competition [20]. It includes a range of puzzles of varying sizes and difficulties that test a wide range of Sokoban solving techniques. The simplest level in the set can be solved with a single move (the only available move instantly solves the level) while the level with the longest optimal path requires a minimum of 1003 moves to solve [24].

We evaluate MCTS-R by comparing its performance against three MCTS configurations, described below, on the Microban test set. We use these to highlight the difficulty with which standard MCTS solves sparse-impact problems, and to compare certain optimisations with our approach. We evaluate all four configurations in time-limited and computation-limited experiments. Additionally, we compare the structure of the search trees generated by all four approaches to show the impact of MCTS-R's compression of the search tree, and consider the differences in the quality of the solutions generated by the two best-performing configurations.

A. Experimental Setup

We compare MCTS-R against other MCTS configurations by studying its performance under two conditions: bounded by wall clock time, and bounded by the number of iterations of the search. In both cases our measure of success is the number of levels from the test set for which a solution of any length was found. We also record the *unique state ratio*, which we define as the number of unique states encountered when building the search tree, divided by the number of nodes in the search tree at the end of the search. This allows us to contrast the repetition of states in standard MCTS with the effect of tree compression in MCTS-R.

We assess the algorithms on both time and iterations because the reversibility checking and compression processes used by MCTS-R are more computationally intensive than standard MCTS. This means that comparing the performance on the same number of iterations would not give a full picture of the differences between the algorithms. Note that if MCTS-R compresses the tree during Selection, it skips to the next iteration without Expansion or Simulation, as the compression may require the Selection process to be restarted. However, we consider an iteration to have taken place, for the purposes of the iteration-bounded experiments.

In addition to MCTS-R, we ran experiments on three MCTS variants. The first configuration is standard MCTS, with no adaptation. The reward for a game state, which the algorithm calculates at the end of the Simulation step, is equal to the game's score – in the case of Sokoban, the score is equal to the number of crates on goal tiles divided by the total number of goal tiles, or the maximum integer value if complete. This reward is based on the GVGAI Competition [20].

In the second configuration, MCTS+N, we add a small novelty component to the reward for the Simulation step. During this step, we record the number of states seen that have not been encountered during the construction of the search tree. We then add a small factor to the reward based on the proportion of states seen in the simulation that are new: $0.9 \times (c/t) + 0.1 \times (n/s)$, where c is the number of covered goal tiles, and t is the total number of goal tiles, n is the number of novel states encountered during simulation (i.e., that do not exist in the search tree) and t is the total number of states encountered during the simulation. This is

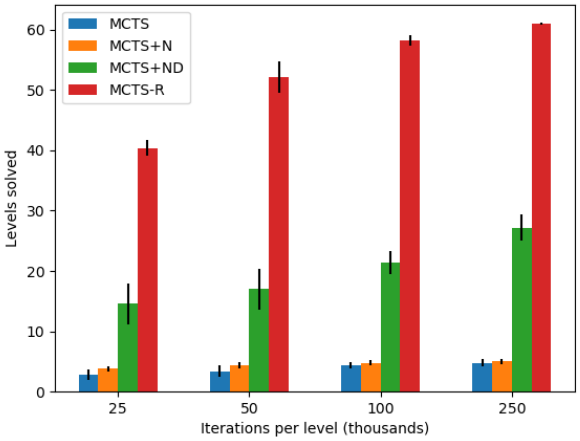

Fig. 3. Iteration-limited run on the Microban test set.

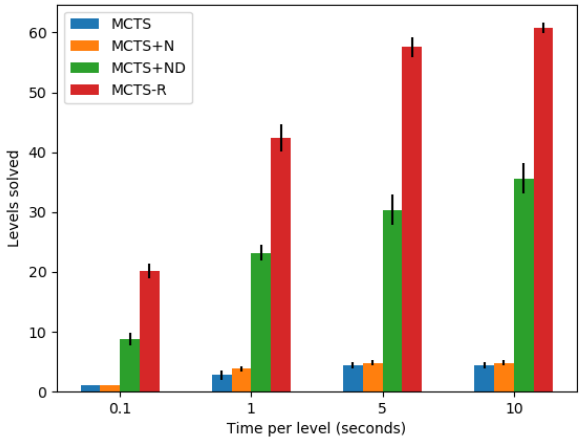

Fig. 4. Time-limited run on the Microban test set.

a common extension made to MCTS algorithms for general game-playing, as it rewards the expansion of search tree nodes that explore new areas of the search space. We would expect this to improve performance in a sparse-impact environment, as it would reward states which avoided repeated actions and explored new parts of the search space.

In the third configuration, denoted MCTS+ND, we augment the Expansion step of MCTS, where new nodes are added to the search tree. If a new node is created during Expansion, but its state already exists somewhere else in the tree, we simply do not add the state to the tree. This means MCTS+ND can never revisit a state it has already encountered. This optimisation is employed in [18] as a potential improvement for solving Sokoban specifically.

All four algorithms - MCTS-R and the three MCTS configurations - are run with a simulation depth capped at 50 moves, and a C value, which is used in the standard UCT evaluation in the Selection step, set to 1. This is recommended for many single-player tasks [22] and we confirmed experimentally that it gave best results for all configurations. Each configuration is an average of five runs. All experiments were implemented in C#, and run on a 2018 Macbook Pro, using a 2.6GHz Intel Core i7 processor, and 16GB of DDR4 RAM.

B. Results

1) Iteration-Limited: Figure 3 shows the results of running all four configurations on the Microban test set, limited by the number of iterations the algorithm is run for. We can see that both MCTS and MCTS+N struggle to solve more than a few cases regardless of computation limits, while MCTS+ND performs far better. However, MCTS-R clearly outperforms all three configurations, solving more than twice as many levels as MCTS+ND in each test.

2) Time-Limited: Figure 4 shows the results of running all configurations on the same test set, limited by wall clock time. We chose a range of time limits, with the shortest being close to a real-time interactive response (0.1 seconds) and the upper bound being 10 seconds. As with iteration-limiting, we can see that MCTS and MCTS+N are unable to solve many levels even as the limit scales. MCTS+ND performs well again, but MCTS-R once again outperforms it, solving 1.8 times as many levels at lower iteration counts, and over double at the highest.

3) Unique State Ratio: Table I shows the ratio of unique states to the size of the final search tree for the four configurations, expressed as an average of all data in each experiment category. To calculate this, during each search process we record the number of unique search nodes added to the tree, and then divide this number by the size of the tree at the end of the search. We can see from table I that MCTS and MCTS+N have ratios below 0.02 for all experiments. A ratio of 0.02 means that fifty nodes must be added to the tree in order to encounter a novel system state.

MCTS+ND always has a ratio of 1, because each state is only ever encountered once, since the algorithm does not allow duplicate states. MCTS-R, however, achieves ratios of higher than 1, managing over seven times the ratio of MCTS+ND in one case, and over 1,000 times higher than standard MCTS. This means that for every node in the final search tree, MCTS-R has encountered seven unique states on average. We believe this efficiency is a factor in the success of the algorithm.

4) Solution Lengths: Finally, we compared the solutions found by both MCTS-R, and the next best-performing configuration, MCTS+ND in the most successful experimental setup: time-limited, 10-second cap. We considered only levels for which both algorithms found a solution, and compared the lengths of the solutions (in Sokoban, a shorter solution is better). We divided the length of MCTS+ND's solution by MCTS-R's solution to derive a relative scale. The average of these scales is 0.93, meaning that, on average, the solutions found by MCTS+ND are slightly shorter than MCTS-R's. The highest and lowest ratios were 1.40 and 0.27 respectively, showing that both algorithms were capable of outperforming in terms of solution quality. In general, MCTS+ND outperforms

System	Setup	Unique States Per Tree Node
MCTS	Iterations	0.0066
MCTS+N	Iterations	0.0092
MCTS+ND	Iterations	1
MCTS-R	Iterations	7.11
MCTS	Time	0.014
MCTS+N	Time	0.017
MCTS+ND	Time	1
MCTS-R	Time	6.33

TABLE I
AVERAGE NUMBER OF UNIQUE STATES ENCOUNTERED PER NODE IN THE FINAL SEARCH TREE, ACROSS ALL EXPERIMENTS.

MCTS-R on levels which involve many 'dead-end' actions which render the level unsolvable, while MCTS-R outperforms on levels with complex optimisations and shortcuts. MCTS+ND's approach of never duplicating a node may help it avoid large areas of 'dead-end' states by only considering them once. We will investigate this as a point of future work.

VI. Conclusions and Future Work

In this paper we introduced *Monte Carlo Tree Search with Reversibility Compression*, or MCTS-R. We showed that standard MCTS approaches perform poorly on sparse-impact problems, where many actions have no permanent impact on the state. By incorporating *reversibility compression* and adapting it to MCTS, MCTS-R can dynamically compress search trees to reduce the significance of reversible actions, with no loss of information.

We showed that MCTS-R outperforms three variants of MCTS on a standard test set of sparse-impact problems, both in wall-clock times and computation required. We also showed that MCTS-R's compression allows it to produce more efficient, compact search trees. In the future we aim to study how transposition tables might enable the benefits of compression to be distributed more broadly across the search tree. We also intend to apply the technique to creative domains such as automated game design. Unlike general game-playing, in which games can be relied upon to be well-formed, automatically designed games are often disfunctional in some regard. We believe that MCTS-R will be effective in rapidly evaluating such games, as it can automatically detect and abstract away uninteresting actions, even with no prior knowledge about the domain.

VII. Acknowledgements

The author wishes to thank the reviewers for detailed feedback. My apologies for not including more of your thoughtful suggestions – I have noted them for the future.

References

[1] Aijun Bai, Siddharth Srivastava, and Stuart Russell. Markovian state and action abstractions for mdps via hierarchical mcts. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, 2016.

[2] Vincent Bloemen. On-the-fly parallel decomposition of strongly connected components. University of Twente, 2015.

[3] C. Browne and F. Maire. Evolutionary game design. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(1):1–16, 2010.

[4] Cameron Browne, Edward Jack Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[5] Vadim Bulitko, Nathan R Sturtevant, Jieshan Lu, and Timothy Yau. Graph abstraction in real-time heuristic search. *Journal of Artifical Intelligence Research*, 30:51–100, 2007.

[6] B. E. Childs, J. H. Brodeur, and L. Kocsis. Transpositions and move groups in monte carlo tree search. In *Proceedings of the IEEE Symposium On Computational Intelligence and Games*, 2008.

[7] Michael Cook and Simon Colton. Redesigning computationally creative systems for continuous creation. In *Proceedings of the Ninth International Conference on Computational Creativity*, 2018.

[8] Michael Cook and Azalea Raad. Hyperstate space graphs for automated game analysis. In *Proceedings of the IEEE Conference on Games*, 2019.

[9] Richard Dearden and Craig Boutilier. Abstraction and approximate decision-theoretic planning. *Artificial Intelligence*, 89(1-2), 1997.

[10] D. Dor and U. Zwick. Sokoban and other motion planning problems. *Computational Geometry*, 13:215–228, 1999.

[11] F. Frydenberg, K. R. Andersen, S. Risi, and J. Togelius. Investigating mcts modifications in general video game playing. In *IEEE Conference on Computational Intelligence and Games*, 2015.

[12] Kalin Gochev, Benjamin J. Cohen, Jonathan Butzke, Alla Safonova, and Maxim Likhachev. Path planning with adaptive dimensionality. In *Proceedings of the Fourth International Symposium on Combinatorial Search*, 2011.

[13] Richard D. Greenblatt, Donald E. Eastlake, and Stephen D. Crocker. The greenblatt chess program. In *Proceedings of the Fall Joint Computer Conference*, 1967.

[14] Daniel Harabor and Alban Grastien. Online graph pruning for pathfinding on grid maps. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, page 11141119. AAAI Press, 2011.

[15] Robert C. Holte, Taieb Mkadmi, Robert M. Zimmer, and Alan J. MacDonald. Speeding up problem solving by abstraction: A graph oriented approach. *Artificial Intelligence*, 85(1-2):321–361, 1996.

[16] Eldane Vieira Junior and Rita Maria Julia. Btt-go: An agent for go that uses a transposition table to reduce the simulations and the supervision in the monte-carlo tree search. In *Proceedings of the Twenty-Seventh International Florida Artificial Intelligence Research Society Conference*, 2014.

[17] Akihiro Kishimoto and Martin Müller. A general solution to the graph history interaction problem. In Deborah L. McGuinness and George Ferguson, editors, *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, pages 644–649. AAAI Press / The MIT Press, 2004.

[18] Fabio Marocchi and Mattia Crippa. Monte carlo tree seach for sokoban. Master's thesis, Politecnico di Milano, 2018.

[19] Luvneesh Mugrai, Fernando Silva, Christoffer Holmgard, and Julian Togelius. Automated playtesting of matching tile games. In *Proceedings of the IEEE Conference on Games*, 2019.

[20] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Simon M. Lucas, and Tom Schaul. General video game ai: Competition, challenges, and opportunities. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI Press, 2016.

[21] Maarten P. D. Schadd, Mark H. M. Winands, Mandy J. W. Tak, and Jos W. H. M. Uiterwijk. Single-player monte-carlo tree search for samegame. *Knowledge-Based Systems*, 34:311, 2012.

[22] Maarten P. D. Schadd, Mark H. M. Winands, H. Jaap van den Herik, Guillaume M. J. B. Chaslot, and Jos W. H. M. Uiterwijk. Single-player monte-carlo tree search. In *Proceedings of the International Conference on Computers and Games*, 2008.

[23] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[24] David W. Skinner. Microban. tinyurl.com/microbanlevels, 2000.

[25] Nathan R. Sturtevant. Memory-efficient abstractions for pathfinding. In *Proceedings of the Third AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press, 2007.