Solving infinite games on trees with back-edges

Aniruddh Gandhi\textsuperscript{1} \hspace{1cm} Bakhadyr Khoussainov\textsuperscript{2} \hspace{1cm} Jiamou Liu\textsuperscript{3}

\textsuperscript{1} Department of Computer Science, University of Auckland
\hspace{0.5cm} Auckland, New Zealand.
\hspace{0.5cm} Email: agan014@aucklanduni.ac.nz

\textsuperscript{2} Department of Computer Science, University of Auckland
\hspace{0.5cm} Auckland, New Zealand.
\hspace{0.5cm} Email: bmk@cs.auckland.ac.nz

\textsuperscript{3} School of Computing and Mathematical Sciences, Auckland University of Technology,
\hspace{0.5cm} Auckland, New Zealand
\hspace{0.5cm} Email: jiamou.liu@aut.ac.nz

Abstract

We study the computational complexity of solving the following problem: Given a game \( G \) played on a finite directed graph \( G \), output all nodes in \( G \) from which a specific player wins the game \( G \). We provide algorithms for solving the above problem when the games have Büchi and parity winning conditions and the graph \( G \) is a tree with back-edges. The running time of the algorithm for Büchi games is \( O(\min\{r \cdot m, \ell + m\}) \) where \( m \) is the number of edges, \( \ell \) is the sum of the distances from the root to all leaves and the parameter \( r \) is bounded by the height of the tree. The algorithm for parity has a running time of \( O(\ell + m) \).

1 Introduction

In the last 10-20 years, there has been an extensive study of infinite games played on directed graphs. These games are natural models for reactive systems (9), concurrent and communication networks (12), and have close interactions with model checking, verification problems, automata and logic (3, 8, 11, 14).

In this paper, we study turned-based games where the players move by the other player, we say that the starting \( 1 \) is the opposite. If Player \( \sigma \) chooses an outgoing edge \((u,v)\) and places the token on node \( v \). The players play the game indefinitely and thus producing a possibly infinite walk in the graph.

The goal of Player 0 is to produce a walk that satisfies the winning condition, while the goal of Player 1 is the opposite. If Player \( \sigma \) wins regardless of the moves by the other player, we say that the starting node \( u \) is a winning position of Player \( \sigma \). An algorithm solves the game if it detects all the winning positions of Player 0. Formally, the winning region problem is defined as follows:

Copyright ©2012, Australian Computer Society, Inc. This paper appeared at the 18th Computing: Australasian Theory Symposium (CATS 2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 128, Julian Mestre, Ed. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

Given a game \( \mathcal{G} \) played on a finite graph \( G \), output all nodes in \( G \) from which Player 0 has a strategy to win the game \( \mathcal{G} \).

This paper studies the algorithms for solving the winning region problem for Büchi and parity games played on trees with back-edges. Given a directed graph \( G \), a Büchi game played on \( G \) specifies a set of target nodes \( T \) in \( G \), and Player 0 wins the game from a node \( u \) if the player has a strategy to visit nodes in \( T \) infinitely often starting from \( u \). It is well-known that the winning region problem for Büchi games can be solved in polynomial time (7, Ch.2). A parity game on \( G \) associates a priority \( p(u) \in \mathbb{N} \) with every node \( u \). Player 0 wins the game from a node \( u \) if the player has a strategy such that the minimum priority amongst all the nodes visited infinitely often in any play starting from \( u \) is even. Though intensively studied, polynomial time algorithms for solving the winning region problem for parity games remain unknown. Parity games are known to be in \( \text{NP} \cap \text{Co-NP} \) but not known to be in \( \text{P} \) (7, Ch.6).

While Büchi games are solved in polynomial time for arbitrary graphs, it is still unclear whether the known algorithms are optimal for a given class of games. This paper concentrates on algorithms for solving the winning region problem for games with Büchi winning conditions played on trees with back-edges. In this paper we will demonstrate that, even for this severely restricted subclass of infinite games, the analysis for the winning region problem can still be non-trivial and that the winning region may be rich in structure. We then apply our analysis for the case of Büchi games to solve the winning region problem for games with the parity winning condition played on trees with back-edges.

The classical algorithm for solving Büchi games uses iterations: at iteration \( i \), the algorithm computes the set of nodes \( U_i \) from which Player 0 has a strategy to visit the target set \( i \) times (see Section 2 for a detailed description of the classical algorithm). It can be shown that the set of winning positions for Player 0 is \( \bigcap_{i \in \mathbb{N}} U_i \) (7). Each iteration of the algorithm takes time \( O(n + m) \), where \( m \) and \( n \) are the number of edges and nodes in \( G \) respectively. The algorithm performs at most \( n \) iterations and hence the running time of the algorithm is \( O(n \cdot (n + m)) \).

The classical algorithm has a seemingly repetitive nature as a node may be processed several times. Hence, it makes sense to carry out a more detailed analysis of Büchi games and see if the classical algorithm can be improved. For instance, the paper (5) investigates the class of graphs with con-
stant out-degrees and shows that solving Büchi games
played on such graphs can be done in $O(n^{2} \log n)$
time. For graphs with unbounded out-degrees, the paper (4)
 presents an algorithm that runs in time $O(n \cdot m \cdot \log \delta(n)/ \log n)$ where $\delta(n)$ is the out-degree of
the game graph. These investigations suggest the
idea of designing more efficient algorithms in specified
classes of graphs such as trees with back-edges.

Trees with back-edges are widely used and studied in
computer science. The paper (6) studies counterex-
amples in model checking whose transition di-
grams are trees with back-edges. Furthermore, as
discussed in (1), they form a natural class of directed
graphs that has directed tree-width 1 and unbounded
entanglement. Also, consider the trees generated by
depth-first search. If the original graph has only tree
edges and back-edges but no cross-edges, then the
algorithms described in this paper can be used. An
other use of trees with back-edges is in $\mu$-calculus
where the syntax graph of a $\mu$-calculus formula is a
tree with back-edges (2). As pointed out in (2) a finite
Kripke structure can be viewed as a tree with back-
edges by performing a partial unraveling of the
structure.

In our analysis, we use the notion of snare to clas-
sify the winning nodes of Player 0 as follows. In-
tuitively, a snare of rank 0 is a subtree from which
Player 0 has a strategy to stay in the subtree forever
and win the game. A snare of rank $i$, $i > 0$, is a
subtree from which Player 1 may choose between two
options: (a) staying in the subtree forever and losing
the game, or (b) going to an $(i - 1)$-snare. We show
that the collection of all snares corresponds exactly to
winning nodes of Player 0. In particular, we present
an efficient algorithm that solves a Büchi game played
on trees with back-edges. The algorithm runs in time
$O(\min(r, m, l + m))$ where $r$ is the largest rank of a
snare (whose value is at most the height of the tree)
and $l$ is the external path length, i.e., sum of the dis-
tances from the root and all leaves, in the underlying
tree.

We then give an algorithm for solving parity games
played on trees with back-edges by reduction to Büchi
games on trees with back-edges and prove the fol-
lowing theorem: any parity game played on trees
with back-edges can be solved in time $O(l + m)$. J.
Obdržálek in his work (13) (Chapter 3) outlines a
proof that parity games played on trees with back-
edges are solvable in polynomial time. The work
does not provide a detailed analysis of the algorithm
but rather concentrates on attacking the problem for
the class of all parity games. The algorithm detects
whether Player 0 wins the game from the root of the
tree and it is claimed that this can be done in time
$O(m)$ (where $m$ is the number of edges in the graph).

We would like to point out that the running time of
any algorithm for solving the winning region problem
is heavily dependent on the data structures and un-
derlying model of computation. In particular, under
the reasonable assumption that trees with back-edges
are encoded as binary strings it can be shown that
$O(m \log(m))$ bits are necessary to encode a tree
with back-edges with $O(m)$ edges. This can be seen as
follows: for a tree $T$, we determine the main branch
starting from the root and always choosing the next
node $v$ such that $v$ has the most number of nodes be-
low it. Now consider a tree $T$ with $O(m)$ edges such
that the main branch has $m/4$ nodes and the first $m/4$
nodes of the main branch have exactly one offbranch-
ing leaf. Below the first $(m/4)^{th}$ nodes of the main
branch, there is a full binary tree with $m/2$ nodes and
$m/4$ leaves. If we now consider the class of trees with
back-edges that can be obtained from $T$ by adding
exactly one back-edge per leaf, it can be seen that
there are $O((m/4)(m/4))$ trees with back-edges in this
class. Hence we need at least $O(m \cdot \log(m))$ bits to
encode a tree with back-edges with $O(m)$ edges. This
fact shows that any algorithm to solve the winning
region problem for games on trees with back-edges
must have a running time of at least $O(m \cdot \log(m))$.
Hence it is not clear how the time bound of $O(m)$
of (13) can be achieved when trees with back-edges are
encoded using binary strings.

Noteworthy among the above observations, the algo-
rit hm of (13) can be modified to run in $O(h \cdot m)$ (where
$h$ is the height of the underlying tree). In this paper
we give an alternative algorithm for solving parity
games on trees with back-edges based on our analy-
sis for Büchi games played on trees with back-edges
which has a running time of $O(\ell + m)$. Note that
since $\ell$ maybe much smaller as compared to $h \cdot m$,
our algorithm performs better than the time bound
of $O(h \cdot m)$ in many cases. Importantly, we clarify the
data structures and model of computation used (see
Section 3) and hence analyze the problem in greater
detail.

Since the worst case performance of our algorithm
for Büchi games is the same as that of the clas-
sical algorithm, we carried out experiments to compare
the actual performance of the two algorithms. The
experiments can be broadly divided into two cate-
gories: 1) comparing the average running times
for games with a small $n$ (number of nodes) and 2)
for large $n$, we compare the running times of the two
algorithms on games whose underlying trees belong
to different classes of randomly generated trees. We
found that our algorithm has significantly better per-
formance than the classical algorithm in both these
categories. Our algorithm outperforms the classical
algorithm by an order of magnitude for even small val-
ues of $n$. The performance gap becomes even clearer
for large values of $n$, where our algorithm again has a
significantly better running time than the classical
algorithm in all the classes of randomly generated trees
used. In fact our algorithm has a linear growth in
running time as compared to the quadratic growth in
running time for the classical algorithm for all the
classes of randomly generated games.

The rest of the paper is organized as follows. Section 2
describes the known algorithm for solving Büchi
games. Section 3 lays out the basic frame-
work and proves a normal form lemma (Lemma 2)
for games played on trees with back-edges. Section 4
introduces the notion of snares and describes the al-
gorithm that uses snares to solve Büchi games played
on trees with back edges. In Section 5 we apply the
algorithm to parity games played on trees with back-
edges. Finally we present experimental results to sup-
port our claims in Section 6.

2 Games Played on Finite Directed Graphs

For background on games played on graphs, see e.g.
(7). A game is a tuple $G = (V_0, V_1, E, \text{Win})$ where
$G = (V_0 \cup V_1, E)$ forms a finite directed graph (called
the underlying graph of $G$), $V_0 \cap V_1 = \emptyset$ and the set
$\text{Win} \subseteq (V_0 \cup V_1)^\omega$. Nodes in the set $V_0$ are said to be
0-nodes and nodes in the set $V_1$ are said to be 1-nodes.
We use $V$ to denote $V_0 \cup V_1$. If $u, v \in V$ then the
set $\{v | (u, v) \in E\}$. The game is played by Player 0
and Player 1 in rounds. Initially, a token is placed on
some initial node $v \in V$. In each round, if the token

\[ V_0 \subseteq V. \]
is placed on a node $u \in V_\sigma$, where $\sigma \in \{0, 1\}$, then Player $\sigma$ selects a node $u' \in E(u)$ and moves the token from $u$ to $u'$. The play continues indefinitely unless the token reaches a node $u$ where $E(u) = \emptyset$. Thus, a play starting from $u$ is a (possibly infinite) sequence of nodes $\pi = v_0 v_1 \ldots$ such that $v_0 = u$ and for every $i \geq 0$, $v_{i+1} \in E(v_i)$. We use $\text{Plays}(G)$ to denote the set of all plays starting from any node in $V$. The winning condition of $G$, denoted by $W_{\text{reach}}$, is a subset of $\text{Plays}(G)$ and Player 0 wins a play $\pi \in \text{Plays}(G)$ if $\pi \in \text{Win}$ and Player 1 wins $\pi$ otherwise. We use $\text{Occ}(\pi)$ to denote the set of nodes that appear in $\pi$ and $\text{Inf}(\pi)$ to denote the set of nodes that appear infinitely often in $\pi$.

A reachability game is a game $G = (V_0, V_1, E, W_{\text{reach}})$. The winning condition $W_{\text{reach}}$ is determined by a set of target nodes $T \subseteq V$ such that

$$W_{\text{reach}} = \{ \pi \in \text{Plays}(G) \mid \text{Occ}(\pi) \cap T \neq \emptyset \}.$$ 

Hence, for convenience, we denote the reachability game by $(V_0, V_1, E, T)$. A Büchi game is a game $G = (V_0, V_1, E, W_{\text{Buchi}})$. As in the case of reachability games, we specify a set of target nodes $T \subseteq V$. Then the Büchi winning condition can be expressed as follows:

$$W_{\text{Buchi}} = \{ \pi \in \text{Plays}(G) \mid \text{Inf}(\pi) \cap T \neq \emptyset \}.$$ 

We use the tuple $(V_0, V_1, E, \rho)$ to denote a parity game.

When playing a game, the players use strategies to determine the next move from the previous moves. Formally, a strategy for Player $\sigma$ (or a $\sigma$-strategy), where $\sigma \in \{0, 1\}$, is a function $f_\sigma : V^* \to V$ such that if $f_\sigma(v_1 v_2 \ldots v_i) = w$ then $(v_i, w) \in E$ (here $V^*$ denotes the set of all finite paths in the graph $G$ with the last node in $V_1$). A play $\pi = v_0 v_1 \ldots$ is consistent with $f_\sigma$ if $v_{i+1} = f_\sigma(v_1 v_2 \ldots v_i)$ whenever $v_i \in V_0$ ($i \geq 0$). A strategy $f_\sigma$ is winning for Player $\sigma$ on $v$ if Player $\sigma$ wins all plays starting from $v$ consistent with $f_\sigma$. If Player $\sigma$ has a winning strategy on $u$, we say Player $\sigma$ wins the game on $u$, or $u$ is a winning position for Player $\sigma$. The winning region, denoted by $W_\sigma$, is the set of all winning positions for Player $\sigma$. Note that $W_0 \cap W_1 = \emptyset$. A strategy $f_\sigma$ for Player $\sigma$ is called memoryless if $f_\sigma(v_1 v_2 \ldots v_i) = f_\sigma(v_i)$ for all $(v_1 v_2 \ldots v_i) \in V^*$.

For the sake of simplicity, we assume $E(u) \neq \emptyset$ for all $u \in V$ in any game $G$. This can be achieved by performing the following whenever $E(u) = \emptyset$: we add two extra vertices $u_1, u_2$ such that $E(u) = u_1, E(u_1) = u_2$ and $E(u_2) = u_1$. In the case of reachability and Büchi games we declare $u_1, u_2 \not\in T$ and for parity games we declare $u_1, u_2 \in T$ and have odd priorities. Note that this does not change the winning region of Player 0 for any of the winning conditions described earlier. Hence we may assume that $\text{Plays}(G) \subseteq V^\omega$.

For the rest of this paper, we will assume that our underlying model of computation is a random access machine (RAM) and that manipulating registers/pointers of logarithmic lengths takes constant time.

A game enjoys determinacy if $W_0 \cup W_1 = V$. The determinacy result in the next theorem is a special case of the well-known Borel determinacy theorem which states all Borel games enjoy determinacy.

**Theorem 1.** (10)(7) Reachability, Büchi and parity games enjoy memoryless determinacy. Moreover, computing $W_0$ and $W_1$ takes time $O(m + n)$ for a reachability game and $O(n \cdot (m + n))$ for a Büchi game, where $m, n$ are respectively the number of edges and nodes in the underlying graph.

By the above theorem, we are justified in restricting ourselves to memoryless strategies for such games and in this paper we shall only consider memoryless strategies. By solving a game, we mean to provide an algorithm that takes as input a game $G$, and outputs all nodes in $W_0$. We briefly describe the classical algorithms that solve reachability and Büchi games.

We first provide the algorithm for solving reachability games. Suppose $G$ is a reachability game. For $Y \subseteq V$, let

$$\text{Pre}(Y) = \{ v \in V_0 \mid \exists u : (v, u) \in E \land u \in Y \}$$

$$\cup \{ v \in V_1 \mid u : (v, u) \in E \implies u \in Y \}.$$ 

The algorithm computes a sequence of sets $T_0, T_1, T_2, \ldots$ where $T_0 = T$, and for $i > 0, T_i = \text{Pre}(T_{i-1}) \cup T_{i-1}$. Since the graph is finite, we have $T_i = T_{i+1}$ for some $i \in N$ and $u$ is a winning position for Player 0 if and only if $u \in T_i$. This algorithm can be implemented to run in $O(m + n)$ by performing a reverse breadth first search (starting from $T_i$) to compute the out-degrees of the nodes followed by a second reverse breadth first search to compute the $T_i$'s exploiting the out-degrees computed previously. The reader is directed to (7, Ch.2) for details.

We refer to this algorithm as the reach algorithm. Let $G$ be the underlying graph of the game. For any set $X \subseteq V$, we use $\text{Reach}_\sigma(X, G)$ to denote the $\sigma$-winning region for the reachability game $(V_0, V_1, E, X)$. In other words, from any node in $\text{Reach}_\sigma(X, G)$, Player $\sigma$ has a strategy that forces any play starting from this node to visit $X$. Hence in the above algorithm we have $T_s = \text{Reach}_0(T, G)$.

We now present the classical algorithm for solving Büchi games. Suppose $G$ is a Büchi game. Compute the sequences of sets $T_0, T_1, \ldots, R_0, R_1, \ldots$ and $U_0, U_1, \ldots$ as follows: Let $T_0 = T$. Suppose $T_i$ is defined for $i > 0 \geq 0$. Set $R_i = \text{Reach}_0(T_i, G)$ and $U_i = V \setminus R_i$. Set $T_{i+1} \leftarrow T_i \setminus \text{Reach}_0(U_i, G)$. Hence we have $T_0 \supseteq T_1 \supseteq T_2 \supseteq \ldots$. The process terminates when we have $T_s = T_{s+1}$ for some $s \in N$. A node $v$ is a winning position for Player 0 if and only if $v \in \text{Reach}_0(T_s, G)$. The algorithm takes time $O(n \cdot (m + n))$. See (7, Ch.2) for details.

### 3 Trees with back-edges

We consider rooted directed trees where all edges are directed away from the root. All terminologies on trees are standard. The ancestor relation on a tree $T$ is denoted by $\leq_T$ and the root is its least element. For $u \leq_T v$, let $\text{Path}[u, v] = \{ x \mid u \leq_T x \leq_T v \}$. The level $\text{lev}(u)$ of a node $u \in V$ is the length of the unique path from the root to $u$. The height $h$ of the tree is $\max\{|\text{lev}(v)| \mid v \in V\}$. The external path length $l$ is $\sum\{|\text{lev}(v)| \mid v \text{ is a leaf in } T\}$.
Definition 1. A directed graph $G = (V, E)$ is a tree with back-edges if its edge relation $E$ can be partitioned into two sets $E^T$ and $E^B$ such that $\forall (u, v) \in E^T$: $u \preceq_T v$ and $\forall (u, v) \in E^B$: $u \prec_T v$. The graph $G = (V, E^T \cup E^B)$ is a rooted directed tree and all edges in $E^B$ are of the form $(v, u)$ where $u \prec_T v$. Edges in $E^B$ are called back-edges. We denote a tree with back-edges by $(V, E^T \cup E^B)$. We refer to leaves of $T$ as leaves of $G$. A Büchi game is played on a tree with back-edges if its underlying graph is a tree with back-edges.

Let $G = (V, E^T \cup E^B)$ be a tree with back-edges. A subtree of $G$ is a subgraph of $G$ that is also a tree. In particular, all the edges of a subtree are from $E^T$ and the root of the subtree is not necessarily the root of $G$. A subtree with back-edges consists of a subtree and all induced back-edges on the subtree. We use $\mathcal{B}$ to denote the class of all Büchi games played on trees with back-edges. A game $G \in \mathcal{B}$ is denoted by the tuple $(V_0, V_1, E^T, E^B, T)$ where $T \subseteq V$ are target nodes. Recall that we may assume without loss of generality that for any node $u \in V$, we have $E^T(u) \cup E^B(u) \neq \emptyset$.

In the rest of the paper we will present our algorithm for solving the winning region problem for games in $\mathcal{B}$. Our claim on the running time of the algorithm depends on the following lemmas and the underlying model of computation. A node $u$ in a game $G \in \mathcal{B}$ is stored as the tuple

$$(p(u), \text{tar}(u), \text{pos}(u), \text{Ch}(u), \text{InBk}(u), \text{OutBk}(u)),$$

where $p(u)$ is a pointer to the parent of $u$, $\text{tar}(u)$ is true if and only if $u \preceq_T p(u)$, $\text{pos}(u) = \sigma$ if and only if $u \in V_\sigma$, $\text{Ch}(u)$ is a list of children of $u$, $\text{InBk}(u)$ is a list of incoming back-edges into $u$, $\text{OutBk}(u)$ is a list of outgoing back-edges from $u$. Note that as pointed out in section 2, our underlying model of computation is a random access machine and manipulating registers (of logarithmic lengths) takes constant time. Hence, accessing $p(u)$, $\text{tar}(u)$, $\text{pos}(u)$ as well as the first elements of $\text{Ch}(u)$, $\text{InBk}(u)$ and $\text{OutBk}(u)$ takes constant time. In the following we define a canonical form for all games $G \in \mathcal{B}$.

Definition 2. A Büchi game $G \in \mathcal{B}$ is reduced if the following conditions hold:

- For all $(u, v) \in E^B$, $u$ is a leaf in the underlying tree with back-edges $(V, E^T \cup E^B)$.
- All target nodes are leaves.
- Each leaf in $(V, E^T \cup E^B)$ has exactly one outgoing back-edge.

The following lemma is easy to see.

Lemma 1. Suppose $G$ is a reduced Büchi game and $\pi$ is a play in $G$. Let $R$ be the set of leaves that are visited infinitely often by $\pi$. Then we have

$$\text{Inf}(\pi) = \bigcup \{|\text{Path}(v, u)| (u, v) \in E^B, u \in R\}.$$

The next lemma reduces solving games in the class $\mathcal{B}$ to solving games that are reduced.

Lemma 2. Given a Büchi game $G = (V_0, V_1, E^T, E^B, T) \in \mathcal{B}$, there exists a reduced game $Rd(G) = (U_0, U_1, E^T, E^B, S)$ such that

- $V \subseteq U$ and $|U| \leq |V| + |E^B|$.
- A node $v \in V$ is winning for Player 0 in $G$ if and only if $v$ is winning for Player 0 in $\text{Rd}(G)$.
- $\text{Rd}(G)$ is constructed from $G$ in time $O(|E^T_1 \cup E^B_1|)$.

Proof. The game $\text{Rd}(G)$ is constructed from $G$ as follows:

1. For each back-edge $(u, v) \in E^B$, add a new leaf $\alpha(u, v)$ and subdivide the edge $(u, v)$ into $(u, \alpha(u, v))$ and $(\alpha(u, v), v)$.
2. $S = \{\alpha(u, v) \mid |\text{Path}(v, u) \cap T| \neq 0\}$.

See Fig. 1 for an example. It is easy to see that $\text{Rd}(G)$ is a reduced game and that $V \subseteq U$ and $|U| \leq |V| + |E^B|$. We now need to prove that a node $v \in V$ is winning for Player 0 in $G$ if and only if $v$ is winning for Player 0 in $\text{Rd}(G)$. The proof for this is quite technical and is omitted due to space restrictions.

The target level of a node $u \in V$ is the number of target nodes that occur on the unique path from the root to $u$. If a target node $u$ is visited, we increase the $k$-value by 1; see Algorithm 1. The algorithm is executed with parameters $(r, 0)$ where $r$ is the root of $(V, E^T \cup E^B)$.

Algorithm 1 AssignLabel$(u, i)$.

1. if $\text{tar}(u)$ then $k(u) \leftarrow i + 1$
2. else $k(u) \leftarrow i$
3. end if
4. for $v \in \text{Ch}(u)$ do
5. $\ell(v) \leftarrow \ell(u) + 1$
6. Run AssignLabel$(v, k(u))$
7. end for

The algorithm then copies the nodes and edges in the tree $(V, E^T_1)$ to $\text{Rd}(G)$. When a back-edge $(u, v)$ is detected, the algorithm creates a new node $\alpha(u, v)$, and connects $u$ (resp. $\alpha(u, v)$) with $\alpha(u, v)$ (resp. $v$). Finally, the node $\alpha(u, v)$ is set as a target in $S$ if $u$ and $v$ have different target levels. Algorithm 1 runs in time $O(|V|)$ because the algorithm visits each node in the tree exactly once. The construction of $\text{Rd}(G)$ takes $O(|E^T_1 \cup E^B_1|)$ time because each edge (tree edge or back-edge) in $G$ is visited exactly once.

4 Solving Büchi games played on trees with back-edges

Our goal is to describe an algorithm that solves a Büchi game played on trees with back-edges. By Lemma 2, it suffices to describe an algorithm that solves reduced Büchi games.

4.1 Snare

Let $G = (V_0, V_1, E^T, E^B, T)$ be a reduced Büchi game. Let $u$ be a leaf in the tree $T = (V, E^T)$. Since $G$ is reduced, we abuse the notation by writing $E^B(u)$ for the unique node $v$ such that $(u, v) \in E^B$. We will often identify a subset $S \subseteq V$ with the subgraph of $G = (V_0 \cup V_1, E^T \cup E^B)$ induced by $S$. Recall
that $W_0$ denotes the 0-winning region of $G$. We now give a refined analysis of the set $W_0$ by introducing the notion of snares. Essentially, we will construct a sequence $S_0 \subseteq S_1 \subseteq S_2 \subseteq \ldots$ of subsets of nodes in $V$ that contain all nodes in $W_0$.

**Definition 3.** For a subset $S \subseteq V$, a snare strategy in $S$ is a strategy for Player 0 such that all plays consistent with the strategy starting from a node in $S$ stay in $S$ forever. A 0-snare is a subtree $S$ of the tree $(V, E_T)$ such that all leaves in $S$ are targets and Player 0 has a snare strategy in $S$.

Note that by definition, Player 0 wins the Büchi game $G$ from any nodes in a 0-snare. On the other hand, there can be winning positions of Player 0 that do not belong to any 0-snares. To capture the entire winning region $W_0$ of Player 0, we inductively define the notion of $i$-snares for all $i \in \mathbb{N}$. Let $S_0$ be the set $\{u \mid u \text{ belongs to a 0-snare in } G\}$, and let $T_0 = T$. For $i > 0$, define the set

$$T_i = \{x \mid E^B(x) \in S_{i-1}\},$$

where the sets $S_1, S_2, \ldots$ are defined inductively as follows.

**Definition 4.** For $i > 0$, an $i$-snare is a subtree $S$ of the tree $(V, E_T)$ such that

- All leaves of $S$ are in $T \cup T_i$. 
- From any node $v \in S$, Player 0 has a snare strategy in $S \cup S_{i-1}$.

We let $S_i$ denote the set of all nodes that are in an $i$-snare. 

Note that the sequence of nodes $S_0, S_1, \ldots$ satisfies that

$$S_0 \subseteq S_1 \subseteq S_2 \subseteq \ldots$$

The snare rank of the node $u \in V$ is $\min\{i \mid u \in S_i\}$. The snare rank of the Büchi game $G$ is the maximum snare rank of the nodes in $G$. From now on we always use $r$ to denote the snare rank of $G$. Note that the definition requires that $G$ is a reduced game. When $G$ is not reduced, the snare rank of $G$ is defined on the game $Rd(G)$. We use the term snare to refer to an $i$-snare for some $i \in \{0, \ldots, r\}$.

As an example, consider the Büchi game shown in Fig. 2. The 0-snares and 1-snare are shown in the figure. Note that the subtree with back-edges rooted at 5 and containing 7, 8, 11, 12 is not a 0-snare. Node 13 is the root of a 1-snare. Note that from 13, Player 1 has two options: 1) move to 15 and lose the game since 16 is a target node or 2) move to 14 and lose the game since the play must move to the 0-snare rooted at 3.

**Proposition 1.** The snare rank $r$ of $G$ is bounded by the height $h$ of the tree $(V, E_T)$. 

Figure 1: Example of a Büchi game played on a tree with back edges and the equivalent reduced game.

Figure 2: Example of a Büchi game with snares shown.
Proof. Let \( H_r \) be a snare of rank \( r \) with root \( u_r \). It is clear from the definition of an \( i \)-snare that there must be a leaf \( x \) in \( H_r \) such that there is a back-edge from \( x \) to some node \( v \in S_{r-1} \). Also note that all nodes in \( \text{Path}[u,r] \) belong to \( H_r \) and therefore \( v \prec u_r \).

Let \( u_{r-1} \) be the root of the snare \( H_{r-1} \) which contains \( v \) (\( u_{r-1} \leq v \prec u_r \)). Since \( u_{r-1} \prec u_r \), we have \( \text{lev}(u_{r-1}) < \text{lev}(u_r) \). We can apply a similar argument to the snare \( H_{r-2} \) to find a snare \( H_{r-2} \) which has a root \( u_{r-2} \prec u_{r-1} \) (\( \text{lev}(u_{r-2}) < \text{lev}(u_{r-1}) \)). In this manner we find a sequence of nodes \( u_{r-1}, u_{r-2}, \ldots \) such that each \( u_i \) is the root of an \( i \)-snare and for each \( i \) we have \( \text{lev}(u_{i+1}) < \text{lev}(u_i) \).

Since the tree \((V,E^T)\) has height \( h \), the value of \( r \) is bounded by \( h \).

The following lemmas reduce the problem of solving a reduced Büchi game to computing snares.

**Lemma 3.** If \( u_0 \in S_i \) for some \( i \in \{0, \ldots, r\} \), then \( u_0 \) is a win.

**Proof.** We prove the lemma by induction on \( i \). Suppose \( u_0 \) belongs to an \( i \)-snare \( S \) for some \( i > 0 \). Let \( f \) be a snare strategy in \( S \) which is consistent with \( f \) either stays in \( S \) forever or goes to an \( (i-1) \)-snare. If all plays starting from \( u_0 \) consistent with \( f \) reach \( S_{i-1} \), then \( u_0 \) is a win by the inductive assumption.

Suppose \( \pi = u_0, u_1, \ldots, u_i \) is a play consistent with \( f \) that does not reach \( S_{i-1} \). Since the game \( G \) is reduced, the play \( \pi \) eventually reaches a leaf node. Let \( u_j \) be the first leaf node visited by \( \pi \). Then \( u_j \in T_i \cup T_i^v \). If \( E^B(u_j) \in S_{i-1} \), then \( u_{j+1} \in S_{i-1} \) which is impossible by assumption. Hence, by definition, \( u_j \) is a target node. Now applying the same argument to the play starting from \( u_{j+1} \), we obtain \( u_j \), which is the second leaf node visited by \( \pi \). Continuing this argument, we obtain a sequence of target nodes \( u_{j+1}, u_{j+2}, \ldots \) of nodes in \( \pi \) where \( j_0 < j_1 < j_2 < \cdots \). Hence \( \pi \) is a winning play for Player 0. This concludes the proof that \( u_0 \in W_0 \).

Now our goal is to show that every node in \( W_0 \) belongs to some snare. We need the following definition:

**Definition 5.** Let \( f \) be a winning strategy for Player 0 on a node \( u \). We define the tree induced by \( f \) on \( u \) as a subtree \( T^f_u = (V^f_u, \leq f) \) of the underlying tree \((V,E^T)\) such that the root of \( T^f_u \) is \( u \) and for every node \( v \prec u \), every \( w \in V^f_u \) whenever

- \( w = f(v) \) for some \( x \in V_0 \cap V^f_u \), or
- \( w \in E^T(x) \) for some \( x \in V_1 \cap V^f_u \).

The edge relation \( E_{u,f} \) is \( E^T \) restricted to \( V^f_u \).

**Lemma 4.** If \( u \in W_0 \) then \( u \in S_i \) for some \( i \in \{0, \ldots, r\} \).

**Proof.** Suppose \( u \) does not belong to any snare. Assume for the sake of contradiction that \( u \in W_0 \). Let \( f \) be the winning strategy for Player 0 starting from \( u \). Consider the tree \( T^f_u \) induced by \( f \) on \( u \). If for all leaves \( v \in T^f_u \) we have \( E^B(v) \in S_i \) for some \( i \in \mathbb{N} \), then by definition \( T^f_u \) is a \((j+1)\)-snare where \( j = \max \{i \mid \text{for every } v \in T^f_u, v \in V_0 \} \). This is in contradiction with the assumption. Therefore, let \( L \) be the non-empty set containing all leaves \( v \) of \( T^f_u \) such that \( E^B(v) \) does not belong to any snare. We define a node \( u_1 \) as follows.

- If all nodes in \( L \) are targets, then there is some \( v \in L \) with \( E^B(v) \prec u \) as otherwise \( T^f_u \) is a \( 0 \)-snare. In this case, let \( u_1 = E^B(v) \).
- Suppose a node \( v \in L \) is not a target. If \( E^B(v) \geq u \), then the path \( v, E^B(v), v, E^B(v), \ldots \) defines a play consistent with \( f \) but is winning for Player 1. Thus \( E^B(v) \prec u \) and we let \( u_1 = E^B(v) \).

Note that in both cases, \( u_1 \prec u \) and \( u_1 \) does not belong to any snare. Also since \( f \) is a winning strategy for Player 0, \( u_1 \) is a winning position for Player 0. Applying the same argument as above, we obtain the sequence \( u_0 < u_1 < u_2 < \cdots \) such that no node in this sequence is in a snare. The sequence is finite and let \( u_2 \) be the last node in it. Any play consistent with \( f \) starting at \( u_2 \) stays in the induced tree \( T^f_{u_2} \) forever.

Therefore if all leaves in \( T^f_{u_k} \) are targets, \( u_k \) belongs to a \( 0 \)-snare which is impossible. Hence let \( y \) be the leaf in \( T^f_{u_k} \) that is not a target. Then the sequence \( u_0, y, E^B(y), y, E^B(y), \ldots \) defines a winning play that is consistent with \( f \) and is winning for Player 1. This is in contradiction with the fact that \( f \) is a winning strategy for Player 0.

Combining Lemma 3 and Lemma 4, we have the following:

**Theorem 2.** For any reduced Büchi game \( G \), the winning region \( W_0 \) of Player 0 coincides with the set \( S_r \), where \( r \) is the snare rank of \( G \).

### 4.2 Finding Snares

Our goal is to present an algorithm that computes all snares in the reduced game \( G \). Let \( T = (V,E^T) \). Recall from Section 2 that for any \( X \subseteq V \), Player 0 has a strategy to force any play into \( X \) from Reach\(_0(X,T) \) by using only tree-edges. For simplicity, we denote Reach\(_0(X,T) \) by Reach\(_0(X) \). Note that if \( S_0 \) then \( v \in \text{Reach}(T) \).

For every node \( v \in \text{Reach}(T) \), we define a value \( b_0(v) \in \mathbb{N} \) inductively as follows:

- If \( v \) is a leaf, let \( b_0(v) = \text{lev}(E^B(v)) \). Notice that \( v \in T \).
- If \( v \) is an internal node and \( v \in V_0 \), let \( b_0(v) = \max \{b_0(u) \mid u \in E^T(v) \cap \text{Reach}(T) \} \).
- If \( v \) is an internal node and \( v \in V_1 \), let \( b_0(v) = \min \{b_0(u) \mid u \in E^T(v) \} \).

Intuitively, the value \( b_0(v) \) represents the level in the tree the game may arrive in if the play starts from \( v \) and goes through one back-edge, when the players adopt the following strategy:

- Player 0 would like to stay as close to the leaves as possible.
- Player 1 would like to stay as close to the root as possible.

For any node \( v \in V \), and a strategy \( f \) for Player 0, let

\[ b_0(f,v) = \min \{\text{lev}(E^B(w)) \mid w \text{ is a leaf in the tree } T^f_u \} \]

Intuitively, \( b_0(f,v) \) is the largest number \( k \) with the following property: the first leaf \( w \) that appears in
any play starting from $v$ and consistent with $f$ has $\text{lev}(E^B(w)) \geq k$. In other words if Player 0 adopts the strategy $f$ starting from $v$, then $b_0(f,v)$ is the closest level to the root that Player 1 can guarantee to move to after following exactly one back-edge. Note that when $v$ is itself a leaf, $b_0(f,v) = \text{lev}(E^B(v))$ for any strategy $f$. For any $X \subseteq V$ and $v \in \text{Reach}(X)$, let $S_{\text{Reach}}(X,v)$ denote the set of all 0-strategies that force any play into $X$ from $v$. The next lemma relates $b_0(v)$ with $b_0(f,v)$ for all $f \in S_{\text{Reach}}(T,v)$.

**Lemma 5.** For every $v \in \text{Reach}(T)$, $b_0(v) = \max\{b_0(f,v) \mid f \in S_{\text{Reach}}(T,v)\}$.

The proof of the above lemma is by induction on the level of $v$.

For every $u \in \text{Reach}(T)$, let

$$S_0(u) = \{v \mid v \geq u \land \forall \omega \in \text{Path}[u,v] : b_0(\omega) \geq \text{lev}(\omega)\}.$$ 

The following lemma provides a way to check if a node belongs to a 0-snare.

**Lemma 6.** For every $v \in \text{Reach}(T)$, $v$ belongs to a 0-snare if and only if $v \in S_0(u)$ for some $u \leq v$ such that $b_0(u) \geq \text{lev}(u)$.

**Proof.** Suppose $v$ belongs to a 0-snare $S$ that is rooted at some node $u$. Let $f$ be the snare strategy in $S$. Since all leaves of $S$ are targets, for all $w \in \text{Path}[u,v]$, $f \in S_{\text{Reach}}(T,w)$, and by definition of a snare strategy, all plays starting from $w$ that are consistent with $f$ will stay in $S$ forever. In particular, we have $b_0(f,w) \geq \text{lev}(u)$. By Lemma 5, $b_0(u) \geq b_0(f,w) \geq \text{lev}(u)$. Furthermore, for all nodes $w \in \text{Path}[u,v]$, $b_0(w) \geq b_0(f,w) \geq \text{lev}(w)$. Hence, $v \in S_0(u)$.

Conversely, let $u \in \text{Reach}(T)$ be such that $b_0(u) \geq \text{lev}(u)$. We prove that the set $S_0(u)$ forms a 0-snare. It is clear that all leaves in $S_0(u)$ are targets. By Lemma 5, for every node $v \in S_0(u)$, there is a strategy $f_v$ such that any leave $w$ in the tree $T_{v}^{f_v}$ is a target and $\text{lev}(E^B(w)) \geq b_0(u)$. Therefore we define a strategy for Player 0 such that $g(v) = f_v(v)$ for all node $v \in V_0 \cap S_0(u)$. Now let $\pi$ be a play starting from some $v \in S_0(u)$ and consistent with $g$. Whenever $\pi$ reaches a leaf $w$, we have $\text{lev}(E^B(w)) \geq \text{lev}(u)$ and thus $E^B(w) \in S_0(u)$. Hence any play starting from $S_0(u)$ and consistent with $g$ will stay in $S_0(u)$ forever. This means $S_0(u)$ is a 0-snare.

Lemma 6 gives us a way to check if a node belongs to a 0-snare. In particular, the following equality holds:

$$S_0 = \bigcup \{S_0(u) \mid u \in \text{Reach}(T) \land b_0(u) \geq \text{lev}(u)\}.$$ 

We now apply our reasoning above to i-snares where $i > 0$. For $i > 0$, recall that $T_i = \{w \mid E^B(w) \in S_{i-1}\}$. Note that a node belongs to an i-snare only if it belongs to $\text{Reach}(T \cup T_i)$. Recall that $h$ is the height of the tree $T$. We inductively define a function $b_i : \text{Reach}(T \cup T_i) \rightarrow \{0, \ldots, h\}$ in the same way as $b_0$ with the following difference: If $v \in T_i$, let $b_i(v) = h$. For any node $v \in V$ and a strategy $f$, let $b_i(f,v) = h$ if all leaves in the tree $T_{f,v}^h$ belong to $T_i$ and let $b_i(f,v) = b_0(f,v)$ otherwise. In other words, $b_i(f,v)$ is the largest number $k \in \{0, \ldots, h\}$ with the following property: the first leaf $w$ that appears in any play starting from $v$ and consistent with $f$ has either $E^B(w) \in S_{i-1}$ or $\text{lev}(E^B(u)) \geq k$. In the same way as the proof of Lemma 5, we can prove the following lemma:

**Lemma 7.** For every $v \in \text{Reach}(T \cup T_i)$, $b_i(v) = \max\{b_i(f,v) \mid f \in S_{\text{Reach}}(T \cup T_i,v)\}$.

For every $u \in \text{Reach}(T \cup T_i)$, let

$$S_i(u) = \{v \geq u \land \forall \omega \in \text{Path}[u,v] : b_i(\omega) \geq \text{lev}(\omega)\}.$$ 

We can prove the next lemma similarly as proving Lemma 6 with every appearance of Lemma 5 replaced by Lemma 7.

**Lemma 8.** For any node $v \in \text{Reach}(T \cup T_i)$, $v$ belongs to an i-snare if and only if $v \in S_i(u)$ for some $u \leq v$ such that $b_i(u) \geq \text{lev}(u)$.

Hence, we obtain the following equality for every $i \in \{0, \ldots, r\}$:

$$S_i = \bigcup \{S_i(u) \mid u \in \text{Reach}(T \cup T_i) \land b_i(u) \geq \text{lev}(u)\}. \quad (1)$$

### 4.3 An algorithm for solving Büchi games on trees with back-edges

Recall that for a tree $T$, the external path length $\ell$ is $\sum \{\text{lev}(v) \mid v \in T\}$. For any game $G$ played on trees with back-edges (not necessarily reduced), by the height and external path length of $G$ we respectively mean the height and external path length of the underlying tree of $G$.

**Theorem 3.** There exists an algorithm that solves any Büchi game $G$ played on trees with back-edges in time $O(\min(r, m, \ell + m))$ where $r$ is the snare rank, $m$ is the number of edges and $\ell$ is the external path length of $G$.

**Proof.** By Lemma 2, we first compute in time $O(m)$ the reduced game $\text{Rd}(G)$. The rest of the algorithm works on $\text{Rd}(G)$, which we simply write as $G$. For $i > 0$, assume $S_{i-1}$ has been computed. By Lemma 8 and (1), Algorithm 2 computes the set $S_i$. By Lemma 3 and Lemma 4, we obtain that $W_0 = S_r$. Hence, to compute the entire winning region $W_0$, it suffices to run $\text{FindSnare}[0](G)$, $\text{FindSnare}[1](G)$, ..., in order. The algorithm terminates after running $\text{FindSnare}[r+1](G)$ where $S_r = S_{r+1}$ (and thus $r$ is the snare rank).

**Algorithm 2 FindSnare[i](G).** (Outline)

1. Compute the set $T_i = \{w \mid E^B(w) \in S_{i-1}\}$.
2. Compute $\text{Reach}(T \cup T_i)$.
3. For all $u \in \text{Reach}(T \cup T_i)$ do:
   4. Compute $b_i(u)$
   5. If $b_i(u) \geq \text{lev}(u)$, compute $S_i(u)$ and add $S_i(u)$ to $S_i$. 

In $\text{FindSnare}[i](G)$, $i \in \{0, \ldots, r+1\}$, we compute $b_i(u)$ for all $u \in \text{Reach}(T \cup T_i)$ in order: we only compute $b_i(u)$ when $b_i(v)$ for all $v \in E_i^1(u) \cap \text{Reach}(T \cup T_i)$ have been computed. When $b_i(u) \geq \text{lev}(u)$, we apply a depth-first search on the subtree rooted at $u$ to compute the set $S_i(u)$. After $S_i(u)$ has been computed, we contract all the nodes in $S_i(u)$ into a meta-node $M_{S_i(u)}$ and redirect edges as follows: any edge $(u,v)$ where $u \in S_i(u)$ and $v \not\in S_i(u)$ is substituted by an edge $(M_{S_i(u)}, v)$ and conversely any edge $(v,u)$ where $v \not\in S_i(u)$ and $u \in S_i(u)$ is substituted by $(v, M_{S_i(u)})$.

Hence, each edge in $G$ is visited a fixed number of times and the $\text{FindSnare}[i](G)$ algorithm takes time $O(m)$. 

119
To further reduce the running time of the algorithm, we maintain a variable \( b(u) \) for every node \( u \) throughout the entire algorithm. When \texttt{FindSnare}(i)(G) is executed, \( b(u) \) will store the value of \( b_i(u) \). During the first iteration (when \texttt{FindSnare}[0] is performed), \( b(u) = b_0(u) \) for all \( u \in \text{Reach}(T) \) and undefined for all other nodes. In the subsequent iterations, we do the following to compute \( b_i(u) \) for \( i > 0 \) and \( u \in \text{Reach}(T \cup \{f\}) \):

1. If \( u \) is a leaf in \( T \), set \( b(u) = h \). If \( u \) is a leaf in \( T \), the value of \( b(u) \) remains unchanged.

2. Then “propagate” the value of \( b(u) \) to ancestors of \( u \) as follows: let \( v \) be the parent of \( u \). If \( v \in V_1 \) and \( b(v) > b(u) \), then set \( b(v) = b(u) \). If \( v \in V_0 \) and \( b(v) < b(u) \), then set \( b(v) = b(u) \). This process continues until we reach a node \( w \prec_T u \) such that \( b(w) \) does not need to be updated or \( w \) is the root.

Hence, at any iteration of the algorithm, we only change the value of \( b(v) \) when \( b(u) \) is changed for some leaf \( u \succ_T v \). Also, for any leaf \( u \), if the value of \( b(u) \) is set to \( h \), it is never changed again. Therefore, the number of times we visit a node \( v \in V \) is at most the number of leaves in the subtree rooted at \( v \). This means that the algorithm runs in time \( \mathcal{O}(\ell + m) \) (since the external path length of \( \text{Rd}(G) \) is at most \( \ell + m \)). By the arguments above, we conclude that the algorithm runs in time \( \mathcal{O}(\min\{r \cdot m, \ell + m\}) \). □

5 Solving parity games played on trees with back-edges

We now apply Theorem 3 to obtain an algorithm for solving parity games on trees with back-edges. Recall the definition of parity games from Section 2: In a parity game \( G \), each node \( u \in V \) is associated with a priority \( \rho(u) \in \mathbb{N} \) and Player 0 wins \( \pi \) if only if \( \inf(\pi) \) is even. Also recall that we may assume that \( \text{Add}(G) \) is an even parity game.

To define the sets \( U_0, U_1, E^1_T \), and \( E^0_T \), we use the same construction as in the proof of Lemma 2. The target set \( T \) in the game \( H \) is defined as

\[
T = \{ (u,v) \mid (u,v) \in E^T_1, \min(\rho(u) \mid x \in \text{Path}[v,u]) \text{ is even} \}
\]

We prove the following claim.

Claim. A node \( u \in V \) is winning for Player 0 in \( G \) if and only if \( u \) it winning for Player 0 in \( H \).

Fix \( u \in V \). Suppose \( u \) is a winning position of Player 0 in \( G \). Let \( f \) be the winning strategy for Player 0 at \( u \). By Theorem 1 we know that \( f \) is a memoryless strategy. Define the strategy \( g \) in the same way as in the proof of Lemma 2. Any play \( \pi \) starting from \( u \) and consistent with \( f \) in \( H \) defines a play \( \pi' \) starting from \( u \) and consistent with \( f \) in \( G \) such that \( \inf(\pi') = \inf(\pi) \cap V \). Since \( f \) is a winning strategy for Player 0, \( \min(x \mid x \in \text{Path}(\pi')) \) is even. Let \( e = \min(x \mid x \in \text{Path}(\pi')) \). There must be a back edge \( (x,y) \in E^0_T \), such that \( x \in \text{Path}(\pi) \) and \( y \in \text{Path}(\pi') \). Hence \( x \prec_T y \) in \( G \). Since \( \pi' \) is winning for Player 0, \( \pi \) is also winning for Player 0.

Conversely, suppose \( u \) is winning position of Player 0 in \( H \). Then \( u \) belongs to a snare by Theorem 2. Let \( i \) be the snare rank of \( u \) and let \( S \) be the i-snare containing \( u \). Let \( f \) be a winning strategy for Player 0 in \( S \cup S_i-1 \) that \( S_i \) denotes all nodes in \( H \) that are in an \((i-1)\)-snare. Define the strategy \( g : V_0 \rightarrow V \) in the same way as in the proof of Lemma 2. Let \( \pi \) be a play consistent with \( g \) starting from \( u \) in \( G \). Our goal is to prove that \( \pi \) is a winning play for Player 0 in \( G \). Suppose for the sake of contradiction that \( \pi \) is winning for Player 1.

Note that \( \rho(u) \) corresponds to a play \( \pi' \) consistent with \( f \) such that \( \inf(\pi') = \inf(\pi) \cap V \). By definition of an i-snare, each leaf in the snare \( S \) is either a target or has a back edge that goes \( S_i-1 \). Assume \( \pi' \) never visits \( S_i-1 \). In this case, all leaves visited by \( \pi' \) are targets. Let \( R \) be the set of leaves visited by \( \pi' \), then by Lemma 1, \( \inf(\pi') = \emptyset \) and Player 1 wins. Is even. Hence \( \min(x \mid x \in \text{Path}(\pi)) \) is also even and \( \pi \) is winning for Player 0. This is in contradiction with the assumption that \( \pi \) is winning for Player 1.

Hence \( \pi' \) (and \( \pi \)) must visit a node \( u_1 <_{\pi'} u \) that belongs an \((i-1)\)-snare. Now apply the same argument on \( u_1 \). Continuing this process, we obtain a sequence of nodes \( u = u_0 > u_1 > u_2 > u_3 > \cdots \) such that for all \( j \in \mathbb{N} \), \( u_{j+1} \) has snare rank strictly smaller than the snare rank of \( u_j \). Contradiction.

Hence the claim is proved.

We use the depth-first search (DFS) algorithm on the underlying tree \( T \) of \( G \). Consider a path \( P \) in \( T \) from the root to a leaf. We represent the length of \( P \) by \( |P| \). We use the algorithm of (17) to preprocess \( P \) in time \( \mathcal{O}(|P|) \), such that for any \( u \in P \) and \( v \prec_T u \), we may find the value of \( \min(x \mid x \in \text{Path}(u,v)) \) in constant time. Then it is clear that preprocessing every such path in \( T \) takes \( \mathcal{O}(\ell) \) time (where \( \ell \) is the external path length of \( G \)) and subsequently finding \( \min(x \mid x \in \text{Path}(u,v)) \) for any node \( u <_{\pi} v \) takes constant time.

Hence for every back edge \( (u,v) \in E^\pi \), \( \min(x \mid x \in \text{Path}(u,v)) \) may be found in constant time after the above preprocessing has been completed. Therefore the algorithm constructs the reduced Büchi game \( H \) as follows:

1. Preprocess every path \( P \) from the root to a leaf using the algorithm of (17).
2. For each back edge \( (u,v) \in E^\pi \), create a new leaf \( \alpha(u,v) \) by subdividing \( (u,v) \). Set \( \alpha(u,v) \) as a target if and only if \( \min(x \mid x \in \text{Path}(u,v)) \) is even (note that this takes constant time now).

The above procedure takes time \( \mathcal{O}(\ell + m) \). □
Proceedings of the Eighteenth Computing: The Australasian Theory Symposium (CATS 2012), Melbourne, Australia

By Lemma 9 and Theorem 3 we obtain the following theorem.

**Theorem 4.** Any parity game $G$ played on trees with back-edges can be solved in time $O(\ell + m)$ where $\ell$ is the external path length of $G$ and $m$ is the number of edges in $G$.

6 Experimental results

In order to compare the performance of our algorithm with that of the classical algorithm, we implemented both the algorithms using the Sage mathematics software system (18). All the experiments were performed on an Intel Core 2 Duo processor (2.4 GHz) with a L2 cache of 4 MB and RAM of 3 GB.

The experiments can be broadly divided into two categories:

1. Average case running time comparison: We systematically enumerate all games on trees with back-edges with order $n \in \mathbb{N}$ nodes. For each game, we compare the running times of the classical algorithm and our algorithm.

2. Running time comparison with random sampling: We considered three classes of rooted trees:
   - (a) rooted trees with unbounded out-degree (denoted by RANUD),
   - (b) rooted binary trees (denoted by RANBT),
   - (c) rooted trees where all internal nodes have only a single child node (denoted by RANDL).

We consider the class RANDL since it is the simplest class of rooted trees. Whenever we refer to a random Büchi game $G$ from one of these classes of trees, we mean that the underlying tree of $G$ is a randomly generated tree from that class. Since for a given $n \in \mathbb{N}$, there is a unique tree of order $n$ from RANDL, we describe how we generate random samples of order $n$ from the classes RANUD and RANBT:

- **RANUD**: We first construct a random free tree $T_{\text{free}}$ of order $n$ by generating a sequence of $(n - 2)$ random integers chosen independently and uniformly from $\{0, 1, \ldots, n - 1\}$ and then applying a reverse Prüfer transformation to this sequence (15). We then randomly select a root from the nodes of $T_{\text{free}}$, hence obtaining rooted tree $T$.

- **RANBT**: We construct a random rooted binary tree by the algorithm of (16).

Given a random rooted tree $T$ from RANUD or RANBT, we generate a random reduced Büchi game $\mathcal{G}$ from $T$ as follows:

- For each leaf $v$ of $T$, we make a random choice of an ancestor $u$ of $v$ and declare a back-edge from $v$ to $u$. In this manner we obtain a random tree with back-edges $T_b$.

- From the nodes of $T_b$, we randomly select nodes of Player 0 and target nodes to obtain a random Büchi game $\mathcal{G}$ on trees with back-edges. Note that $\mathcal{G}$ is reduced.

Given a rooted tree $T$ from RANDL, for every node $v$ of $T$ we randomly choose an ancestor $u$ of $v$ and add a back-edge from $v$ to $u$. Then we randomly choose nodes of Player 0 and target nodes as before to obtain a random game $\mathcal{G}$. We then use the procedure described in Lemma 2 to convert $\mathcal{G}$ to a reduced game.

In our experiment, we generated random Büchi games from the three classes described above and then compared the running times of the classical algorithm and our algorithm.

**Results:**

**Average case running time comparison:** Figure 3 (bottom right) shows the average running time of the classical algorithm and our algorithm for games of size $n \in \{5, 6, \ldots, 13\}$. It is clear that our algorithm performs better than the classical algorithm in all cases. Moreover as $n$ increases, the difference in average case running time between the two algorithms increases. This is particularly evident for $n = 13$, where our algorithm performs an order of magnitude better than the classical algorithm.

The picture becomes even clearer when we analyze the results of the experiments with random sampling. In order to enable a more convenient comparison between the two algorithms we have scaled down the running times of the classical algorithm by a factor of $10^2$ for all experiments with random sampling. The sizes of the random games are in the range $n \in \{100, \ldots, 10000\}$.

**Random games from RANUD:** Figure 3 (top left) shows the graph comparing the running time of the classical algorithm versus our algorithm for random Büchi games from the class RANUD. The sample sizes are as follows: $10^3$ random games for $n \in \{100, \ldots, 2000\}$, $10^4$ random games for $n \in \{4000, \ldots, 6000\}$, $5 \cdot 10^4$ games for $n = 8000$ and $4 \cdot 10^4$ games for $n = 10000$.

As the graph shows, the classical algorithm exhibits quadratic growth in running time whereas our algorithm has linear growth in running time (with respect to $n$). This is better than the worst case bound of $O(\min\{r \cdot m, l\})$ mentioned in Theorem 3.

**Random games from RANBT:** Figure 3 (top right) shows the graph comparing the running time of the classical algorithm versus our algorithm for random Büchi games from the class RANBT. The sample sizes are as follows: $10^5$ games for $n \in \{100, \ldots, 2000\}$ and $10^4$ games for $n \in \{4000, 10000\}$. Again the classical algorithm shows a quadratic growth in running time as compared to our algorithm which has a linear growth in running time (w.r.t. $n$) which is better than the worst case bound of $O(\min\{r \cdot m, l\})$.

**Random games from RANDL:** Figure 3 (bottom left) shows the graph comparing the running time of the classical algorithm versus our algorithm for random Büchi games from the class RANDL. The sample sizes are identical to the RANBT case. Here also our algorithm shows a linear growth as compared to the quadratic growth shown by the classical algorithm.

Hence the experiments demonstrate that our algorithm outperforms the classical algorithm in all the three classes. In particular, for $n > 4000$, our algorithm performs two orders of magnitude better than the classical algorithm for all three classes of Büchi games. The experiments clearly show that, in practice, not only our algorithm is much more efficient asymptotically but it also performs better for games with small number of nodes on the set of trees with back-edges.
Figure 3: The top left graph shows the comparison of algorithms for RANUD, the top right graph for RANBT and the bottom left for RANDL (the dashed lines represent the classical algorithm). The bottom right table compares the average running times of the two algorithms. Note that the running time of the classical algorithm has been scaled down by $10^2$ in the graphs.

References


