A Dynamic Algorithm for Reachability Games Played on Trees

Bakhadyr Khoussainov[1], Jiamou Liu[2], Imran Khaliq[2]

Department of Computer Science, University of Auckland, New Zealand
[1]bmk@cs.auckland.ac.nz
[2]{jliu036,ikha020}@aucklanduni.ac.nz

Abstract. Our goal is to start the investigation of dynamic algorithms for solving games that are played on finite graphs. The dynamic game determinacy problem calls for finding efficient algorithms that decide the winner of the game when the underlying graph undergoes repeated modifications. In this paper, we focus on turn-based reachability games. We provide an algorithm that solves the dynamic reachability game problem on trees. The amortized time complexity of our algorithm is $O(\log n)$, where n is the number of nodes in the current graph.

1 Introduction

We start to investigate dynamic algorithms for solving games that are played on finite graphs. Games played on graphs, with reachability, Büchi, Muller, Streett, parity and similar type of winning conditions, have recently attracted a great attention due to connections with model checking and verification problems, automata and logic [6][11][13][17]. Given one of these games, *to solve the game* means to design an (efficient) algorithm that tells us from which nodes a given player wins the game. Polynomial time algorithms exist to solve some of these games, while efficient algorithms for other games remain unknown. For example, on a graph with n nodes and m edges, the reachability game problem is in $O(n + m)$ and is PTIME-complete [8], and Büchi games are in $O(n \cdot m)$ [1]. Parity games are known to be in NP$\cap$ Co-NP but not known to be in P.

An algorithm for solving the games is *static* if the games remain unchanged over time. We pose the *dynamic game determinacy problem*:

> We would like to maintain the graph of the game that undergoes a sequence of update and query operations in such a way that facilitates an efficient solution of the current game.

Contrary to the static case, the dynamic determinacy problem takes as input a game $\mathcal{G}$ and a (finite or infinite) sequence $\alpha_1, \alpha_2, \alpha_3, \ldots$ of *update* or *query* operations. The goal is to optimize the average running time per operation over a worst-case sequence of operations. This is known as the *amortized running time* of the operations.

There has recently been increasing interest in dynamic graph algorithms (See, for example, [4][5]). The dynamic reachability problem on graphs have been investigated in a series of papers by King [9], Demetrescu and Italiano [3], Roditty [14] and Roditty and Zwick [15][16]. In [16], it is shown that for directed graphs with m edges and n nodes, there is a dynamic algorithm for the reachability problem which has an amortized update time of $O(m + n \log n)$ and a worst-case query time of $O(n)$. This paper extends this line of research to dynamic reachability game algorithms. In the setting of games, for a given directed graph G and a player σ, a set of nodes T is *reachable* from a node u in G means that there is a strategy for player σ such that starting from u, all paths produced by player σ following that strategy reach T, regardless of the actions of the opponent. Hence, the dynamic reachability game problem can be viewed as a generalization of the dynamic reachability problem for graphs.

We now describe two-person *reachability games* played on directed finite graphs. The two players are *Player 0* and *Player 1*. The *arena* $\mathcal{A}$ of the game is a directed graph (V_0, V_1, E), where V_0 is a finite set of *0-nodes*, V_1 is a finite set of *1-nodes* disjoint from V_0, and $E \subseteq V_0 \times V_1 \cup V_1 \times V_0$ is the edge relation. We use V to denote $V_0 \cup V_1$. A *reachability game* $\mathcal{G}$ is a pair $(\mathcal{A}, T)$ where $\mathcal{A}$ is the arena and $T \subseteq V$ is the set of *target nodes* for Player 0.

The players start by placing a token on some *initial node* $v \in V$ and then move the token in rounds. At each round, the token is moved along an edge by respecting the direction of the edge. If the token is placed at $u \in V_\sigma$, where $\sigma \in \{0, 1\}$, then Player σ moves the token from u to a v such that $(u, v) \in E$. The play stops when the token reaches a node with no out-going edge or a target node. Otherwise, the play continues forever. Formally, a play is a (finite or infinite) sequence $\pi = v_0 \; v_1 \; v_2 \; \ldots$ such that $(v_i, v_{i+1}) \in E$ for all i. Player 0 *wins the play* π if π is finite and the last node in π is in T. Otherwise, Player 1 wins the play.

In this paper we provide an algorithm that solves the dynamic reachability game played on trees. We investigate the *amortized time complexity* of the algorithm. We concentrate on trees because: (1) Trees are simple data structures, and the study of dynamic algorithms on trees is the first step towards the dynamic game determinacy problem. (2) Even in the case of trees the techniques one needs to employ is non-trivial. (3) The amortized time analysis for the dynamic reachability game problem on graphs, in general case, is an interesting hard problem. (4) Finally, we give a satisfactory solution to the problem on trees. We show that the amortized time complexity of our algorithm is of order $O(\log n)$, where n is the number of nodes on the tree. The space complexity of our algorithm is $O(n)$.

2 A Static Algorithm for Reachability Games

Let $\mathcal{G} = (\mathcal{A}, T)$ be a reachability game. A *(memoryless) strategy* for Player σ is a partial function $f_\sigma : V_\sigma \to V_{1-\sigma}$. A play $\pi = v_0 \; v_1 \; ...$ *conforms* f_σ if $v_{i+1} = f_\sigma(v_i)$ whenever $v_i \in V_\sigma$ and f_σ is defined on v_i for all i. All strategies in this paper are memoryless. A *winning strategy for Player σ from v* is a strategy f_σ such that Player σ wins all plays starting from v that conform f_σ. A node u is a *winning position* for Player σ, if Player σ has a winning strategy from u. The *σ-winning region*, denoted W_σ, is the set of all winning positions for Player σ. Note that the winning regions are defined for memoryless strategies. A game *enjoys memoryless determinacy* if the regions W_0 and W_1 partition V.

Theorem 1 (Reachability game determinacy[7]). *Reachability games enjoy memoryless determinacy. Moreover, there is an algorithm that computes W_0 and W_1 in time $O(n+m)$.*

Proof. For $Y \subseteq V$, set $\text{Pre}(Y) = \{v \in V_0 \mid \exists u[(v,u) \in E \wedge u \in Y]\} \cup \{v \in V_1 \mid \forall u[(v,u) \in E \rightarrow u \in Y]\}$. Define a sequence $T_0, T_1, ...$ such that $T_0 = T$, and for $i > 0$, $T_i = \text{Pre}(T_{i-1}) \cup T_{i-1}$. There is an s such that $T_s = T_{s+1}$. We say a node u has *rank* r, $r \geq 0$, if $u \in T_r - T_{r-1}$. A node u has infinite rank if $u \notin T_s$. Once checks that a node $u \in W_0$ if and only if u has a finite rank. Computing W_0 takes $O(n+m)$ time. □

3 Dynamic Reachability Game Problem: A Set-Up

As mentioned above, the dynamic game determinacy problem takes as input a reachability game $\mathcal{G} = (\mathcal{A}, T)$ and a sequence $\alpha_1, \alpha_2, \ldots$ of *update* and *query* operations. The operations produce the sequence of games $\mathcal{G}_0, \mathcal{G}_1, \ldots$ such that $\mathcal{G}_i$ is obtained from $\mathcal{G}_{i-1}$ by applying the operation α_i. A dynamic algorithm should solve the game $\mathcal{G}_i$ for each i. We use the notation

$$\mathcal{A}_i = (V_{0,i} \cup V_{1,i}, E_i),\ V_i,\ T_i,\ W_{\sigma,i}$$

to denote the arena, the set of nodes, the target set and σ-winning region for $\mathcal{G}_i$. We define the following six *update operations* and one *query* operation:

1. *InsertNode*(u,i,j) operation, where $i,j \in \{0,1\}$, creates a new node u in V_i. Set u as a target if $j=1$ and not a target if $j=0$.
2. *DeleteNode*(u) deletes $u \in V$ where u is *isolated*, i.e., with no incoming or outgoing edge.
3. *InsertEdge*(u,v) operation inserts an edge from u to v.
4. *DeleteEdge*(u,v) operation deletes the edge from u to v.
5. *SetTarget*(u) operation sets node u as a target.
6. *UnsetTarget*(u) operation sets node u not as a target.
7. *Query(u)* operation returns **true** if $u \in W_0$ and **false** if $u \in W_1$

By Theorem 1, each node of V_i belongs to either $W_{0,i}$ or $W_{1,i}$.

Definition 1. *A node u is in* state σ *if $u \in W_\sigma$. The node u* changes its state *at stage $i+1$, if u is moved either from $W_{0,i}$ to $W_{1,i+1}$ or from $W_{1,i}$ to $W_{0,i+1}$.*

Using the *static algorithm* from Theorem 1, one produces two *lazy* dynamic algorithms for reachability games. The first algorithm runs the static algorithm after each update and therefore query takes constant time at each stage. The second algorithm modifies the game graph after each update operation *without* re-computing the winning positions, but the algorithm runs the static algorithm for the *Query(u)* operation. In this way, the update operations take constant time, but *Query(u)* takes the same time as the static algorithm. The amortized time complexity in both algorithms is the same as the static algorithm.

4 Reachability Game Played on Trees

This section is the main technical contribution of the paper. All trees are directed, and we assume that the reader is familiar with basic terminology for trees. A *forest* consists of pairwise disjoint trees. Since the underlying tree of the game undergos changes, the game will in fact be played on forests. We however still say that a *reachability game* $\mathcal{G}$ *is played on trees* if its arena is a forest F. We describe a fully dynamic algorithm for reachability games played on trees.

A forest F is implemented as a doubly linked list $\mathrm{List}(F)$ of nodes. A node u is represented by the tuple $(p(u), \mathrm{pos}(u), \mathrm{tar}(u))$ where $p(u)$ is a pointer to the parent of u ($p(u) = \mathrm{null}$ if u is a root), $\mathrm{pos}(u) = \sigma$ if $u \in V_\sigma$ and a boolean variable $\mathrm{tar}(u) = \mathbf{true}$ iff u is a target.

Our algorithm supports all the operations listed in Section 3. At stage s, the algorithm maintains a forest F_s obtained from F_{s-1} after performing an operation. We briefly discuss the operations and their implementations:

– Inputs of the update and query operation are given as pointers to their representatives in the linked list $\mathrm{List}(F_s)$.
– The operations *InsertNode* and *DeleteNode* are performed in constant time. The *InsertNode*(u, i, j) operation links the last node in $\mathrm{List}(F_s)$ to a new node u. The *DeleteNode*(u) operation deletes u from $\mathrm{List}(F_s)$.
– All other update operations and the query operation have amortized time $O(\log n)$, where n is the number of nodes in V.
– The *InsertEdge*(u, v) operation is performed only when v is the root of a tree not containing u. *InsertEdge*(u, v) links the trees containing u and v. *DeleteEdge*(u, v) does the opposite by splitting the tree containing u and v into two trees. One contains u and the other has v as its root.

4.1 Splay Trees

We describe *splay trees* (see [10] for details) which we will use in our algorithm. The *splay trees* form a dynamic data structure for maintaining elements drawn from a totally ordered domain D. Elements in D are arranged in a collection P_D of splay trees, each of which is identified by the root element.

– *Splay*(A, u): Reorganize the splay tree A so that u is at the root if $u \in A$.

- *Join*(A, B): Join two splay trees $A, B \in P_D$, where each element in A is less than each element in B, into one tree.
- *Split*(A, u): Split the splay tree $A \in P_D$ into two new splay trees

$$\text{Above}(u) = \{x \in A \mid x > u\} \text{ and } \text{Below}(u) = \{x \in A \mid x \leq u\}.$$

- *Max*(A)/*Min*(A): Returns the Max/Min element in $A \in P_D$.

Theorem 2 (splay trees[12]). *For the splay trees on P_D, the amortized time of the operations above is $O(\log n)$, where n is the cardinality of D.* □

4.2 Dynamic Path Partition

Recall that a node u *is in state* σ at stage s if $u \in W_{\sigma,s}$. Denote the state of u at stage s by $\text{State}_s(u)$. The update operations may change the state of u. This change may trigger a series of state changes on the ancestors of u. The state of u *does not change* if no update operation is applied to a descendant of u. We need to have a *ChangeState*(u) algorithm which carries out the necessary updates when the state of u is changed. The *ChangeState(u)* algorithm will not be executed alone, but will rather be a subroutine of other update operations.

One may propose a naive *ChangeState*(u) algorithm as follows. We hold for each node v its current state. When *ChangeState*(u) is called, it first changes the state of u, then checks if the parent $p(u)$ of u needs to change its state. If so, we changes the state of $p(u)$, and checks if the parent of $p(u)$ needs to change its state, etc. This algorithm takes $O(n)$ amortized time. Our goal is to improve this time bound. For this, we introduce *dynamic path partition* explained below.

Let $x <_F y$ denote the fact that x is an ancestor of y in forest F. Set $Path(x, y) = \{z \mid x \leq_F z \leq_F y\}$.

Definition 2. *A* path partition *of a forest F is a collection P_F of sets such that P_F partitions the set of nodes in F and each set in P_F is of the form $Path(x, y)$ for some $x, y \in F$.*

Nodes in $Path(x, y)$ are linearly ordered by $<_F$. Call the element of P_F that contains u the *block* of u. A block is *homogeneous* if all its elements have the same state. A path partition P_F is *homogeneous* if each block in P_F is homogeneous. For any u in F_s, set $\nu_s(u) = |\{v \mid (u, v) \in E_s \wedge \text{State}_s(u) = \text{State}_s(v)\}|$.

Definition 3. *A node u in F_s is* stable *at stage s if $u \in T_s$ or for some $\sigma \in \{0, 1\}$, $u \in V_{\sigma,s} \cap W_{\sigma,s}$ and $\nu_s(u) \geq 2$. We use Z_s to denote the set of stable nodes at stage s.*

The following lemma shows how state changes influence non-stable nodes in homogeneous fragments of blocks. The proof follows from the definitions.

Lemma 1. *Suppose x is stable, $x \leq_{F_s} y$, $Path(x, y)$ is homogeneous and there is no stable node in $\{z \mid x <_{F_s} z \leq_{F_s} y\}$. If y changes state at stage $s+1$ then the nodes in the set $\{z \mid x <_{F_s} z \leq_{F_s} y\}$ are precisely those nodes that need to change their states.* □

Definition 4. *A path partition of F is* stable *if whenever a node u is stable, it is the $\leq_F$-maximum element in its own block.*

From this definition, for a stable path partition, if u is stable, all elements $x \neq u$ in the block of u are *not* stable. An example of a stable and homogeneous partition is the trivial partition consisting of singletons.

At stage s, the algorithm maintains two data structures, one is the linked list $\text{List}(F_s)$ as described above, and the other is the path partition P_{F_s}. Each node u in $\text{List}(F_s)$ has an extra pointer to its representative in P_{F_s}. The path partition is maintained to be homogeneous and stable. Denote the block of u at stage s by $B_s(u)$. To obtain logarithmic amortized time, each $B_s(u)$ is represented by a splay tree.

4.3 *ChangeState*(u) Algorithm

The *ChangeState*(u) algorithm carries out two tasks. One is that it changes the states of all the necessary nodes of the underlying forest once u changes its state. The second is that it changes the path partition by preserving its homogeneity and stability properties. For the ease of notations, in this subsection we do not use the subscripts s for the forest F_s and the target set T_s. We write F for F_s and T for T_s.

We explain our *ChangeState(u)* algorithm informally. Suppose u changes its state at stage $s+1$. The algorithm defines a sequence of nodes $x_1 >_F x_2 >_F \ldots$ where $x_1 = u$. For $i \geq 1$, the algorithm splits $B_s(x_i)$ into two disjoint sets $\text{Above}(x_i)$ and $\text{Below}(x_i)$ and temporarily sets $B_{s+1}(x_i) = \text{Below}(x_i)$. By homogeneity, all nodes in $\text{Below}(x_i)$ have the same state at stage s. Change the state of all nodes in $B_{s+1}(x_i)$ (this can be done by Lemma 1) and join the current two blocks containing u and $B_{s+1}(x_i)$ into one. If $\min\{B_{s+1}(x_i)\}$ is the root, stop the process. Otherwise, consider the parent of $\min\{B_{s+1}(x_i)\}$ which is $w_i = p(\min\{B_{s+1}(x_i)\})$. If $\text{State}_s(w_i) \neq \text{State}_s(x_i)$ or $w_i \in Z_s$, stop the loop, do not define x_{i+1} and $B_{s+1}(u)$ is now determined. Otherwise, set $x_{i+1} = w_i$ and repeat the above process for x_{i+1}. Consider the last w_i in the process described above that did not go into the block of u. If $w_i \notin Z_s$ and it becomes stable after the change, split $B_s(w_i)$ into $\text{Above}(w_i)$ and $\text{Below}(w_i)$ and declare that $w_i \in Z_{s+1}$. These all determine the new partition at stage $s+1$.

Algorithm 1 implements the *ChangeState(u)* procedure. The current block of v is denoted by $B(v)$. Elements of $B(v)$ are stored in a splay tree with order $\leq_F$. With the root of each splay tree $B(v)$ the variable $q(B(v)) \in \{0,1\}$ is associated to denote the current state of nodes in $B(v)$. The **while** loop computes the sequence $x_1, x_2, \ldots$ and $w_1, w_2, \ldots$ described above using the variables x and w. The boolean variable ChangeNext decides if the **while** loop is active. The boolean variable $\text{Stable}(v)$ indicates whether v is stable. With each stable node v, the variable $n(v)$ equals $\nu(v)$ at the given stage.

The next two lemmas imply that the path partition obtained after the execution of *ChangeState(u)* remains homogeneous and stable.

Algorithm 1 ChangeState(u).

```
1: ChangeNext ← **true**; $x \leftarrow u$.
2: **while** ChangeNext **do**
3:    Split($B(x), x$); $q(B(x)) \leftarrow 1 - q(B(x))$; Join($B(u), B(x)$).
4:    **if** $p(\min\{B(x)\})$ = null **then** Stop the process. **end if**
5:    $w \leftarrow p(\min\{B(x)\})$.
6:    **if** Stable($w$) $\vee\ q(B(w)) = q(B(u))$ **then** ChangeNext ← **false**. **end if**
7:    $x \leftarrow w$.
8: **end while**
9: Run **UpdateStable**($x, u$)
```

Algorithm 2 UpdateStable(x, u).

```
1: **if** Stable($x$) **then**
2:    $n(x) \leftarrow [q(B(u)) = q(B(x))?\ n(x) + 1 : n(x) - 1]$.
3:    **if** $n(x) < 2 \wedge \neg$ tar($x$) **then** Stable($x$) ← **false**. **end if**
4: **else**
5:    Stable($x$) ← **true**; Split($B(x), x$); $n(x) \leftarrow 2$.
6: **end if**
```

Lemma 2. $B_{s+1}(u) = \{z\ |State_s(z) \neq State_{s+1}(z)\}$.

Proof. If $z = u$ then State$_s(z) \neq$ State$_{s+1}(z)$. Assume $z \neq u$ and $z \in B_{s+1}(u)$. Let x be the $\leq_F$-maximum node in $B_s(z)$. By definition of $B_{s+1}(u)$, $x \notin Z_s$ and State$_s(x)$ = State$_s(u)$. Therefore by Lemma 1 and the construction of the algorithm, $B_{s+1}(u) \subseteq \{z\ |$State$_s(z) \neq$ State$_{s+1}(z)\}$.

Conversely, if State$_s(z) \neq$ State$_{s+1}(z)$, z must be an ancestor of u. Let x be the $\leq_F$-minimum node in $B_{s+1}(u)$. If x is the root, $\{z\ |$State$_s(z) \neq$ State$_{s+1}(z)\} \subseteq B_{s+1}(u)$. If x is not the root, either $p(x) \in Z_s$ or State$_s(p(x))$ = State$_{s+1}(u)$. Again by Lemma 1 and description of the algorithm, State$_s(x)$ = State$_{s+1}(x)$ and thus $\{z\ |$State$_s(z) \neq$ State$_{s+1}(z)\} \subseteq B_{s+1}(u)$. □

Lemma 3. *Suppose $v \notin Z_s$. The node $v \in Z_{s+1}$ if and only if v is the parent of the $\leq_F$-least node that changes its state at stage $s + 1$.*

Proof. Suppose for simplicity that $v \in V_{0,s}$. The case when $v \in V_{1,s}$ can be proved in a similar way. Suppose one of v's children, say v_0, is the $\leq_F$-least node that changes its state at stage $s + 1$. If State$_s(v) = 1$, then all of its children are in $W_{1,s}$. Thus State$_{s+1}(v_0)$=0, and v should also changes to state 0. This contradicts with the $\leq_F$-minimality of v_1. Therefore $v \in W_{0,s}$. Since $v \notin Z_s$, exactly one of its child, say v_1 is in $W_{0,s}$. If $v_1 = v_0$, then none of v's children is in $W_{0,s+1}$ and State$_s(v) \neq$ State$_{s+1}(v)$. Therefore $v_1 \neq v_0$. Thus at stage $s + 1$, v has exactly two children in $W_{0,s+1}$ and $v \in Z_{s+1}$.

On the other hand, suppose $v \in Z_{s+1}$. This means that, $v \in W_{0,s+1}$ and v has two children v_0, v_1 in $W_{0,s+1}$. Note that for any x, at most one child of x may change state at any given stage. Therefore, at most one of v_0 and v_1, say v_1, is in $W_{0,s}$. Hence $v \in W_{0,s}$, and State$_s(v)$ =State$_{s+1}(v)$. Therefore v_0 is the $\leq_F$-least node that changes state at stage $s + 1$. □

4.4 Update and Query operations

We describe the update and query operations. The query operation takes a parameter u and returns $q(B(u))$, which is the state variable associated with the root of the splay tree representing $B(u)$. We use an extra variable $c(u)$ to denote the current number of children of u.

In principle, the algorithms for operations *InsertEdge*(u, v), *DeleteEdge*(u, v), *SetTarget*(u) and *UnsetTarget*(u) perform the following three tasks. (1) Firstly, it carries out the update operation on u and v. (2) Secondly, it calls *ChangeState*(u) in the case when u needs to change state. (3) Lastly, it updates $c(z), n(z)$ and *Stable*(z) for each z. Task (1) is straightforward by changing the values of $p(v)$ and $\text{tar}(u)$. Task (2) (3) can be done by using a fixed number of **if** statements, each with a fixed boolean condition involving comparisons on the variables. We illustrate this using the *InsertEdge*(u, v) operation as follows.

Algorithm 3 InsertEdge(u, v).

```
1: $p(v) \leftarrow u$; $c(u) \leftarrow c(u) + 1$.
2: if $\neg\ \text{tar}(u) \wedge q(B(u)) \neq q(B(v)) \wedge (c(u) = 1$ or $q(B(u)) \neq \text{Pos}(u))$ then
3:     Run ChangeState$(u)$.
4: else if $\text{Stable}(u) \wedge q(B(u)) = q(B(v))$ then
5:     $n(u) \leftarrow n(u) + 1$.
6: else if $\neg\text{Stable}(u) \wedge c(u) > 1 \wedge q(B(u)) = q(B(v)) = \text{Pos}(u)$ then
7:     Stable$(u) \leftarrow$ true; Split$(B(u), u)$; $n(u) \leftarrow 2$.
8: end if
```

The *InsertEdge*(u, v) operation is described in Algorithm 3. Suppose the path partition on F_s is homogeneous and stable, and

1. $q(B(z)) = \text{State}_s(z)$.
2. $n(z) = |\{z' \mid (z, z') \in E_s \wedge \text{State}_s(z) = \text{State}_s(z')\}|$.
3. $c(z) = |\{z' \mid (z, z') \in E_s\}|$.
4. Stable(z) is *true* if and only if $z \in Z_s$.

Suppose the edge (u, v) is inserted at stage $s + 1$. We prove the following two lemmas which imply the correctness of the algorithm.

Lemma 4. ChangeState(u) *is called in the* InsertEdge(u, v) *algorithm if and only if* $State_s(u) \neq State_{s+1}(u)$.

Proof. Note that *ChangeState*(u) is called if and only if the condition for the **if** statement at Line 2 of Algorithm 3 holds. We prove one direction of the lemma, the other direction is straightforward.

Suppose $\text{State}_s(u) \neq \text{State}_{s+1}(u)$. It must be that $u \notin T_s$ and $\text{State}_s(u) \neq \text{State}_s(v)$. Suppose further that u is not a leaf at stage s. If $u \in V_{0,s} \cap W_{0,s}$, there is a child w of u which is also in $W_{0,s}$. This means that $\text{State}_s(u) = \text{State}_{s+1}(u)$. Similarly, one may conclude that u does not change state if $u \in V_{1,s} \cap W_{1,s}$.

Therefore $u \in (V_{0,s} \cap W_{1,s}) \cup (V_{1,s} \cap W_{0,s})$. Therefore the condition for the **if** statement at Algorithm 3 Line 2 holds. $\square$

Lemma 5. *Stable*(u) *is* **true** *at stage* $s+1$ *if and only if* $u \in Z_{s+1}$.

Proof. It is easy to see that if $u \in Z_s$ then $u \in Z_{s+1}$ and Stable(u) is set to **true** at stage $s+1$. Suppose $u \notin Z_s$. Note that Stable(u) is set to **true** at stage $s+1$ if and only if Line 6 in Algorithm 3 is reached and the condition for the **if** statement at this line holds, if and only if u is not a leaf at stage s, State$_s(u)$ = State$_{s+1}(u)$ =State$_s(v)$ and $u \in V_{\sigma,s} \cap W_{\sigma,s}$ for some $\sigma \in \{0,1\}$.

If u is not a leaf, State$_s(u)$ = State$_{s+1}(u)$ =State$_s(v)$ and $u \in V_{\sigma,s} \cap W_{\sigma,s}$ for some $\sigma \in \{0,1\}$, then there is a child w of u in $W_{\sigma,s}$ and thus $u \in Z_{s+1}$. On the other hand, suppose $u \in Z_{s+1}$. If u changes state, then $u \in V_{\sigma,s} \cap W_{1-\sigma,s}$ for some $\sigma \in \{0,1\}$. This means all children of u are in $W_{1-\sigma,s}$ and $u \notin Z_{s+1}$. Therefore it must be that u did not change state. By definition of a stable node, u is not a leaf, State$_s(u)$ =State$_s(v)$ and $u \in V_{\sigma,s} \cap W_{\sigma,s}$. $\square$

Algorithm 4 DeleteEdge(u,v).

```
1: $p(v) \leftarrow$ null; $c(u) \leftarrow c(u) - 1$.
2: if $B(u) = B(v)$ then Split$(B(u), u)$. end if
3: if $\neg$Stable$(u) \wedge q(B(u)) = q(B(v)) \wedge [(c(u) = 0 \wedge q(B(u)) = 0) \vee (q(B(u)) =$ Pos$(u))]$ then
4:     Run ChangeState$(u)$.
5: end if
6: if Stable$(u) \wedge q(B(u)) = q(B(v))$ then
7:     $n(u) \leftarrow n(u) - 1$.
8:     if $n(u) < 2$ then Stable$(u) \leftarrow$ false. end if
9: end if
```

The *DeleteEdge*(u,v) operation is described in Algorithm 4. Suppose that the edge (u,v) is deleted at stage $s+1$. We also have the following two lemmas which imply the correctness of the algorithm. The proofs are similar in spirit to the proofs of Lemma 4 and Lemma 5.

Lemma 6. ChangeState(u) *is called in the* DeleteEdge(u,v) *algorithm if and only if State*$_s(u) \neq$ *State*$_{s+1}(u)$. $\square$

Lemma 7. *Stable*(u) *is* **true** *at stage* $s+1$ *if and only if* $u \in Z_{s+1}$. $\square$

The *SetTarget*(u) and *UnsetTarget*(u) operations are described in Algorithm 5 and Algorithm 6, respectively.

4.5 Correctness

The next lemma implies the correctness of the algorithms 1-6.

Algorithm 5 SetTarget(u).

```
1: if $c(u) = 0$ then $n(u) = 0$.
2: else if $\text{Pos}(u) \neq q(B(u))$ then $n(u) \leftarrow [\text{Pos}(u) = 0?\ 0 :\ c(u)]$.
3: else if $\text{Stable}(u)$ then $n(u) \leftarrow [\text{Pos}(u) = 0?\ n(u) \colon c(u) - n(u)]$.
4: else $n(u) \leftarrow [\text{Pos}(u) = 0?\ 1 :\ c(n) - 1]$. end if
5: $\text{tar}(u) \leftarrow$ true; $\text{Stable}(u) \leftarrow$true; $\text{Split}(B(u), u)$.
6: if $q(B(u)) = 1$ then Run ChangeState($u$). end if
```

Algorithm 6 UnsetTarget(u).

```
1: if $(\text{Pos}(u) = 0 \wedge n(u) = 0) \vee (\text{Pos}(u) = 1 \wedge n(u) < c(u))$ then
2:     Run ChangeState($u$); $n(u) \leftarrow c(u) - n(u)$.
3: end if
4: $\text{Stable}(u) \leftarrow [(n(u) > 1 \wedge \text{Pos}(u) = q(B(u)))?$ true: false].
5: $\text{tar}(u) \leftarrow$ false.
```

Lemma 8. *At each stage s, $State_s(z) = q(B(z))$ for all node $z \in F_s$.*

Proof. The proof proceeds by induction on s. For simplicity, we assume that the initial forest F_0 contains only isolated nodes. We set for each node z, $p(z) =$ null, $c(z) = n(z) = 0$, $B(z) = \{z\}$, $\text{Stable}(z) =$ **true** if and only if $q(B(z)) = 0$ if and only if $\text{tar}(z) =$ **true**. For the case when F_0 is an arbitrary forest, we may assume that the variables $c(z), n(z), B(z)$, $\text{Stable}(z)$ and $q(B(z))$ has been pre-set to their respect values as described in the previous subsection. We use $\gamma_s(u)$ to denote the number of children of u in stage s and recall that $\nu_s(u)$ is the number of u's children in the same state as u. The lemma follows from the fact that the following six inductive assumptions are preserved at each stage s.

(1) The set $\{B(v) \mid v \in V_{0,s} \cup V_{1,s}\}$ forms a homogeneous path partition of F_s.
(2) For each z, $c(z) = \gamma_s(z)$.
(3) For each z, $n(z) = \nu_s(z)$ whenever z is stable.
(4) For each z, $\text{Stable}(z)$ is set to True if and only if z is a stable node.
(5) The path partition $\{B(v) \mid v \in V\}$ is stable.
(6) For each z, $\text{State}_s(z) = q(B(z))$. □

5 Complexity

We analyze the amortized complexity of our algorithm. Each *InsertEdge*(u, v), *DeleteEdge*(u, v), *SetTarget*(u), and *UnsetTarget*(u) algorithm runs at most once the *ChangeState*(u) algorithm, a fixed number of splay tree operations, and a fixed number of other low-level operations such as pointer manipulations and comparisons. By Theorem 2, each splay tree operation takes amortized time $O(\log n)$ where n is the number of nodes in the forest. Therefore, the amortized time complexity for these operations is $O(\log n)$ plus the amortized time taken by the *ChangeState*(u) algorithm.

We now focus on the amortized time complexity of the *ChangeState*(u) algorithm. The algorithm runs in iterations. At each iteration, it processes the current block of nodes $B(x)$ and examines the parent w of the $\leq_F$-least node in $B(x)$. If w does not exist or does not need to change state, the algorithm stops and calls *UpdateStable*(w, x); otherwise, the algorithm sets x to w and starts another iteration. Each iteration in the algorithm and the *UpdateStable*(w, x) algorithm both run a fixed number of splay tree operations and therefore takes amortized time $O(\log n)$. Therefore the algorithm takes time $O(\tau \log n + \log n)$ to execute a sequence of k update operations, where τ is the number of iterations of *ChangeState*(u). We prove the following lemma which implies the $O(\log n)$ amortized time complexity of the above update operations.

Lemma 9. *For any sequence of k update operations, the total number of iterations ran by the* ChangeState(u) *algorithm is $O(k)$.*

Proof. Given a forest $F = (V, E)$ and path partition P_F. For a node $u \in V$, let $B(u)$ denote the block of u. Define $E_F^P \subseteq P_F^2$ such that for all $u, v \in V$

$$(B(u), B(v)) \in E_F^P \text{ if and only if } (\exists w \in B(u))(\exists w' \in B(v))(w, w') \in E$$

The pair (P_F, E_F^P) also forms a forest, which we call the *partition forest* of P_F.

We prove the lemma using the *accounting method* (see [2]). At each stage s, we define the *credit function* $\rho_s : P_{F_s} \to \mathbb{N}$. For block $B \in P_{F_0}$, let $\rho_0(B)$ be the number of children of B in P_{F_0}. At each stage $s + 1$, the credit function ρ_{s+1} is obtained from ρ_s with the following requirements:

- We create for each execution of *InsertEdge*(u, v) and *ChangeState*(u) an *amortized cost* of 1. This cost contributes towards the *credit* $\rho_s(B(u))$. Note that we can create amortized cost at most 2 at each stage.
- For each iteration of *ChangeState*(u), we take out 1 from $\rho_s(B)$ of some $B \in P_{F_s}$.

Let t_s be the total number of iterations ran at stage s. Our goal is to define ρ_s in such a way that for any $s > 0$,

$$\sum_{B \in P_{F_s}} \rho_s(B) \leq \sum_{B \in P_{F_{s-1}}} \rho_{s-1}(B) + 2 - t_s$$

and $\rho_s(B) \geq 0$ for any $B \in P_{F_s}$. Note that the existence of such a credit function ρ_s implies for any k,

$$\sum_{s=1}^{k} t_s \leq 2k \in O(k).$$

We define our desired credit function ρ_s by preserving, for every stage s, the following invariant:

$$(\forall B \in P_{F_s}) \ \ \rho_s(B) = |\ \{B' \in P_{F_s} \mid (B, B') \in E_{F_s}^P\}\ |.$$

The detailed procedure for defining ρ_s is omitted due to space restriction. $\square$

Theorem 3. *There exists a fully dynamic algorithm to solve reachability games played on trees which supports* $\mathrm{InsertNode}(u, i, j)$ *and* $\mathrm{DeleteNode}(u)$ *in constant time, and* $\mathrm{InsertEdge}(u, v)$, $\mathrm{DeleteEdge}(u, v)$, $\mathrm{SetTarget}(u)$, $\mathrm{UnsetTarget}(u)$ *and* $\mathrm{Query}(u)$ *in amortized* $O(\log n)$*-time where* n *is the number of nodes in the forest. The space complexity of the algorithm is* $O(n)$.

Proof. By Lemma 9, the update operations have amortized time complexity $O(\log n)$. The splay tree data structures take space $O(n)$ at each stage. □

References

1. Chatterjee, K., Henzinger, T., Piterman, N.: Algorithms for büchi games. In: Proceedings of the 3rd Workshop of Games in Design and Verification (GDV'06). (2006)
2. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to algorithms, Second Edition. MIT Press and McGraw-Hill. (2001)
3. Demetrescu, C., and Italiano, G.: Fully dynamic transitive closure: Breaking through the $O(n^2)$ barrier. In: Proceedings of FOCS'00, pp. 381-389. (2000)
4. Demetrescu, C., and Italiano, G.: A new approach to dynamic all pairs shortest paths. In: Proceedings of STOC'03, pp. 159-166. (2003)
5. Eppstein, D., Galil, Z., Italiano, G.: Dynamic graph algorithms. In: Algorithms and Theoretical Computing Handbook, M. Atallah, ed., CRC Press. (1999)
6. Grädel, E.: Model checking games. In: Proceedings of WOLLIC'02, vol.67 of Electronic Notes in Theoretical Computer Science. Elsevier. (2002)
7. Grädel, E., Thomas, W., Wilke, T.: Automata, logics, and infinite games. LNCS 2500, Springer, Heidelberg. (2002)
8. Immerman, N.: Number of quantifiers is better than number of tape cells. In: Journal of Computer and System Sciences, 22:384-406. (1981)
9. King, V.: Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In: Proceedings of FOCS'99, pp 81-91. (1999)
10. Kozen, D.: The design and analysis of algorithms(Monographs in computer science), Springer-Verlag, New York. (1992)
11. Murawski, A.: Reachability games and game semantics: Comparing nondeterministic programs. In: Proceedings of LICS'08, pp. 353-363. (2008)
12. Tarjan, R.: Data structures and network algorithms, vol 44 of Regional Conference Series in Applied Mathematics. SIAM (1983)
13. Radmacher, F., Thomas, W.: A game theoretic approach to the analysis of dynamic networks. In: Proceedings of the 1st Workshop on Verification of Adaptive Systems, VerAS 2007, vol 200(2) of Electronic Notes in Theoretical Computer Science, pp. 21-37. Kaiserslautern, Germany (2007)
14. Rodity, L.: A faster and simpler fully dynamic transitive closure. In: Proceedings of SODA'03, pp. 401-412. (2003)
15. Roditty, L., Zwick, U.: Improved dynamic reachability algorithm for directed graphs. In: Proceedings of FOCS'02, pp. 679-689. (2002)
16. Roditty, L., Zwick, U.: A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In: Proceedings of 36th ACM Symposium on Theory of Computing (STOC'04), pp. 184-191. (2004)
17. Thomas, W.: Infinite games and verification. In: Proceedings of the International Conference on Computer Aided Verification CAV'02, LNCS 2404:58-64. (2002)