CDMTCS Research Report Series

Update Games and Update Networks

Michael J. Dinneen
Bakhadyr Khoussainov
Department of Computer Science
University of Auckland

CDMTCS-105
June 1999

Centre for Discrete Mathematics and
Theoretical Computer Science

Update Games and Update Networks

Michael J. Dinneen and Bakhadyr Khoussainov

Department of Computer Science,
University of Auckland, Auckland, New Zealand
{mjd,bmk}@cs.auckland.ac.nz

Abstract

In this paper we model infinite processes with finite configurations as infinite games over finite graphs. We investigate those games, called *update games*, in which each configuration occurs an infinite number of times during a two-person play. We also present an efficient polynomial-time algorithm (and partial characterization) for deciding if a graph is an *update network*.

1 Introduction

Many real-world systems can be viewed as infinite duration processes with finite states. Several examples can be found in computer operating systems, air traffic control systems, banking systems, and the on-going maintenance of communication networks. A functioning system has to be robust (e.g., an operating system should not crash regardless of what the user does). A termination of any of these systems can be thought of as a failure. Thus we need an infinite duration model to study properties of such systems. In practice these systems have only a finite number of states (e.g., a banking system has a finite number of customers, assets, etc.).

The operation of each system over time enters only a finite number of states and produces an infinite sequence of states, called a *run-time sequence*. Since the number of states is finite, some of the states, called *persistent states*, appear infinitely often in the run-time sequence. The success of a run-time sequence is determined by whether or not the collection of persistent states satisfies certain specifications. Thus, we can view the run-time sequences as plays of a two-player game where one player, called the *Survivor*, tries to ensure that persistent states satisfy some property and the other player, called the *Adversary*, does not.

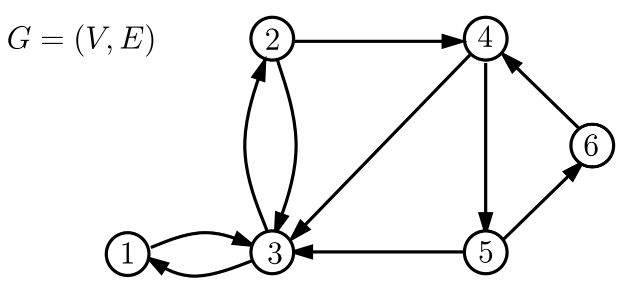
Our proposed model for an infinite duration system is based on a finite (directed) graph. The vertices of the graph represent the states of the system and the edges (arcs) correspond to the legal state changes, called *moves* (or *transitions*), of the system.

Definition 1. An *infinite duration game* $\mathcal{G}$ is a graph $G = (V, E)$, set W of subsets of V, and two players (the Survivor and the Adversary). A member of W is called a *winning set*. A *configuration* of a game is a pair of the form $(v, \text{Survivor})$ or $(v, \text{Adversary})$ for $v \in V$.

The game rules allow configuration moves from (w, X) to (w', X') such that $(w, w') \in E$ and $X \neq X'$. Each *play* of an infinite duration game is a sequence of configurations $(v_0, X_0), (v_1, X_1), \ldots, (v_i, X_i), \ldots$ such that the game rules are followed. We call a finite prefix sequence of a play a *history*. We say that a vertex v is *visited* in the play if configuration (v, X) occurs in the history. Note that either the Survivor or the Adversary may begin the play. The Survivor wins a play if the persistent vertices of the play is a winning set of W, otherwise the Adversary wins. A *strategy* for a player X_i of a game is a function from play histories $(v_0, X_0), \ldots, (v_i, X_i)$ to configurations (v_{i+1}, X_{i+1}) such that the move from (v_i, X_i) to (v_{i+1}, X_{i+1}) is a game rule.

A given strategy for a player X may either win or lose a game when starting at an *initial configuration* (v_0, X_0), where $v_0 \in V$ and X_0 is either player. A player's *winning strategy* for an initial configuration is one that wins no matter what the other player does.

Example 2. In Figure 1 we present a game $\mathcal{G} = (G, W)$. As an example of a winning strategy for the Survivor consider the initial configuration $(4, \text{Adversary})$. If the Adversary moves to vertex 3 then the Survivor simply moves to vertex 1 and the game repeats between those two vertices (which is a winning set). On the other hand, if the Adversary moves to vertex 5, the Survivor moves to vertex 6 forcing the Adversary to move to 4, which is then controlled by the Survivor. The Survivor attempts to force the vertex set $\{4, 5, 6\}$ into a persistent set by moving to vertex 5. If the Adversary tries to move to 3 from 5 then the Survivor is allowed to change its mind and force $\{1, 3\}$ as the persistent set and win. Thus, the Adversary looses no matter what choice is made at vertex 5.



$V = \{1, 2, 3, 4, 5, 6\}$

$E = \{(1, 3), (2, 3), (2, 4), (3, 1), (3, 2), (4, 3), (4, 5), (5, 3), (5, 6), (6, 4)\}$

$W = \{\{1, 3\}, \{2, 3, 4, 5\}, \{4, 5, 6\}\}$

Figure 1: Example of an infinite duration game.

We end this section with a few related references. Previous work on two-player infinite duration games on finite bipartite graphs is presented in the paper by McNaughton [1] and extended by Nerode *et al.* [2]. Also several earlier papers that deal with finite duration games on automata and graphs have appeared (e.g., see [3, 4]).

2 Update Games

We now model a natural communication network problem. Suppose we have data stored on each node of a network and we want to continuously update all nodes with consistent data. For instance, we are interested in addressing redundancy issues in distributed databases. Often one requirement is to share key information between all nodes of the distributed database. We can do this by having a data packet of current information continuously go through all nodes of the network. This is essentially an infinite duration game where the Survivor's objective is to achieve a winning set equal to all the nodes of the network. This game is formally defined as follows:

Definition 3. An *update game* is an infinite duration game $\mathcal{G} = (G, W)$ with the singleton winning set $W = \{V\}$. An *update network* is the underlying graph G of an update game where the Survivor has a winning strategy for each initial configuration.

Sometimes we will talk about a graph G being an update game without mentioning the winning set, since it is understood that $W = \{V\}$.

Example 4. The graph displayed below in Figure 2 is an update network. Notice that the vertices of out-degree 2 and 3 leads only to odd-length cycles so that the Survivor and the Adversary control the vertex every other round. The Survivor can use its opportunities to visit all vertices of the graph.

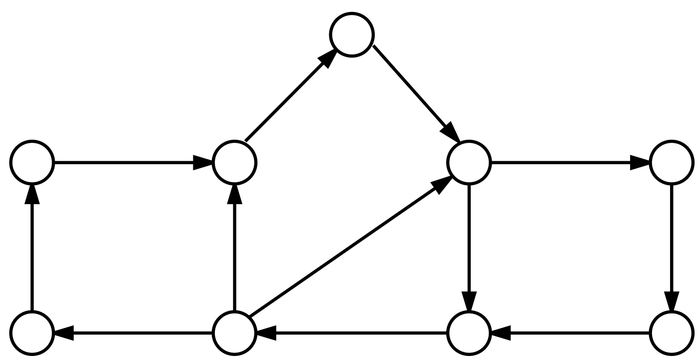


Figure 2: A simple example of an update game which is an update network.

3 Bipartite Update Networks

We first study a special class of update games on bipartite graphs. Here we restrict the domain of graphs to bipartite graphs where the vertices V of each graph can be partitioned into two disjoint sets A and S such that all edges are directed from A to S or from S to A. We also stipulate that each vertex has an out-going edge (i.e., this ensures that every play is of infinite duration). By convention, we assume that the Survivor moves from S and the Adversary moves from A. In essence the vertices are *owned* by the two players of the game.

Definition 5. A *bipartite update network* is bipartite graph $(V = A \cup S, E)$ that can be used as an update game in which the Survivor has a winning strategy to visit every vertex of V infinitely often from every initial configuration. (That is, the Survivor can force the persistent set of vertices to be V.)

Note that there are only $|V|$ game configurations where each vertex v determines an unique configuration depending on whether v is in S or A.

We can easily characterize those bipartite update networks with only one Survivor vertex. These are the bipartite graphs where out-degree$(s) = |A|$ for the single Survivor vertex s. We now derive several properties for all bipartite update networks.

Lemma 6. If $(V = A \cup S, E)$ is a bipartite update network then for every vertex $s \in S$ there exists at least one $a \in A$ such that $(a, s) \in E$ and out-degree$(a) = 1$

Proof. The idea is to show that if there exists a vertex s that does not satisfy the statement of the lemma then the Adversary can always avoid visiting s. Let $A_s = \{a \mid (a, s) \in E\}$ and assume out-degree$(a) > 1$ for all $a \in A_s$. The Adversary has the following winning strategy. If the play history ends in configuration $(a, \text{Adversary})$ then since out-degree$(a) > 1$ the Adversary moves to $(s', \text{Survivor})$, where $s' \neq s$ and $(a, s') \in E$. This contradicts the assumption of lemma. □

For the following results let $(A \cup S, E)$ be a bipartite update game B. For any Survivor vertex s define

$$\text{Forced}(s) = \{a \mid \text{out-degree}(a) = 1 \text{ and } (a, s) \in E\},$$

which denotes the set of Adversary vertices that are 'forced' to move to s.

Lemma 7. If B is a bipartite update network such that $|S| > 1$ then for every $s \in S$ there exists an $s' \neq s$ and an $a \in \text{Forced}(s)$, such that (s', a, s) is a directed path.

Proof. If B has more than one vertex in S then there must be a strategy for the Survivor to create a play history to visit vertex s from some other vertex $s' \in S$. To do this we need a forced Adversary vertex a (of A) in the neighborhood of s'. There exists such a vertex a by Lemma 6. □

Definition 8. Given a bipartite graph $(S \cup A, E)$ a *forced cycle* is a (simple) cycle $(a_k, s_k, \ldots, a_2, s_2, a_1, s_1)$ for $a_i \in \text{Forced}(s_i)$ and $s_i \in S$. Note that forced cycles have even length.

We now present our penultimate ingredient that will be used to characterize bipartite update networks.

Lemma 9. If B is a bipartite update network such that $|S| > 1$ then there exists a forced cycle of length at least 4.

Proof. Take $s_1 \in S$. From Lemma 7 there exists a path (s_2, a_1, s_1) in B such that $s_2 \neq s_1$ and $a_1 \in \text{Forced}(s_1)$. Now for s_2 we apply the lemma again to get a path (s_3, a_2, s_2) in B such that $s_3 \neq s_2$ and $a_2 \in \text{Forced}(s_2)$. If $s_3 = s_1$ we are done. Otherwise repeat Lemma 7 for vertex s_3. If $s_4 \in \{s_1, s_2\}$ we are done. Otherwise repeat the lemma for s_4. Eventually $s_i \in \{s_1, s_2, \ldots, s_{i-2}\}$ since B is finite. □

Note if B does not have a forced cycle of length at least 4 then either $|S| = 1$ or B is not a bipartite update network. That is, if $|S| > 1$ then the Adversary has a strategy to not visit a vertex s_2 of S whenever the play begins at some different vertex s_1 of S. We now present a method that helps us decide if a bipartite game is a bipartite update network.

Lemma 10. If $B = (S \cup A, E)$ is a bipartite update game with a forced cycle of length at least 4 then we can construct a bipartite update game $B' = (S' \cup A', E')$ with $|S'| < |S|$ such that B is a bipartite update network if and only if B' is one.

Proof. We construct B' as follows. Let $C = (a_k, s_k, \dots, a_2, s_2, a_1, s_1)$ be a forced cycle in B of length at least 4. For new vertices a and s let

$$S' = (S \setminus \{s_1, s_2, \dots, s_k\}) \cup \{s\} \quad \text{and} \quad A' = (A \setminus \{a_1, a_2, \dots, a_k\}) \cup \{a\}.$$

Let

$$\begin{aligned} E' \; = \; & E(B \setminus \{s_1, a_1, \dots, s_k, a_k\}) \cup \\ & \{(s, a') \mid a' \in A' \text{ and } (s_i, a') \in E, \text{ for some } i \le k\} \cup \\ & \{(a', s) \mid a' \in A' \text{ and } (a', s_i) \in E, \text{ for some } i \le k\} \cup \\ & \{(s', a) \mid s' \in S' \text{ and } (s', a_i) \in E, \text{ for some } i \le k\} \cup \{(a, s), (s, a)\}. \end{aligned}$$

We now show that if B' is an update network then B is also an update network. We first define the natural mapping p from vertices of B onto vertices of B' by

$$\begin{array}{ll} p(v) = v & \text{if } v \notin C \\ p(v) = a & \text{if } v \in C \cap S \\ p(v) = s & \text{if } v \in C \cap A. \end{array}$$

Then any play history of B is mapped, via the function $p(v) = v'$, onto a play history of B'. Consider a play history $v_0, v_1, \dots, v_n$ of B that starts at vertex v_0. Let f' be a winning strategy for the Survivor when the game begins at vertex v'_0. We use the mapping p to construct the Survivor's strategy f in game B by the following two cases. Case $v'_n = s$. The strategy is to extend the play (in B) by visiting all the vertices of the cycle C at least once. If $f'(v'_0, \dots, v'_n) = a'$ where $a' \ne a$ we find a $s_i \in C$ such that $(s_i, a') \in E$ then extend the play again with a' as the last move. Otherwise $f'(v'_0, \dots, v'_n) = a$ and the play is extended by picking an $a_k \in C$ such that $(v_n, a_k) \in E$. Case $v'_n \ne s$. If $f'(v'_0, \dots, v'_n) = a' \ne a$ then then f will also move to a'. Otherwise $a' \in C$ and the play is extended by picking an $a_k \in C$ such that $(v_n, a_k) \in E$.

It is not hard to see that f is a winning strategy for the Survivor in game B whenever f' is a winning strategy in B'.

We now show that if B is an update network then B' is also an update network. Take any vertex v'_0 from B'. We show that there is a winning strategy for the Survivor starting at v'_0. Fix any vertex v_0 such that $p(v_0) = v'_0$. We will keep a correspondence between positions v_i of a play on B with positions v'_i of a play on B'. We now simulate the winning strategy f on B starting at v_0. The strategy for the initial play history $v'_0 \in S'$ is $f'(v'_0) = p(f(v_0))$ except for the case $v'_0 = s$ (or $v'_0 = a$, which is a forced move

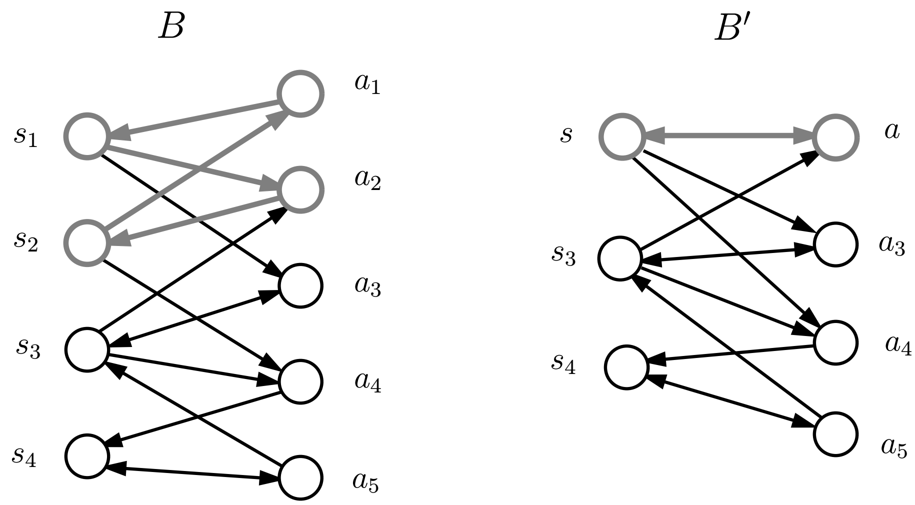


Figure 3: Showing the bipartite update game reduction of Lemma 10.

to $v_1' = s$). In this exceptional case the Survivor's initial strategy is to move directly to any $a' \neq a$ and replace f with the strategy starting at a'. Now let $v_0', v_1', \ldots, v_n'$ be any play history of B' that occurs after the initial play as dictated above. We define a strategy for f' when v_n' is in S' by studying two cases.

Case $v_n' = s$. Consider the previous vertex $v_i' = a' \neq a$ of the play history that is in A' and $v_{i+1}' = s$. Thus in the game on B, the Adversary from a' elects to move to some s_i in C. Since respect to the actual game played on B' the Adversary has less power, we can pick, without loss of generality, that it moved to s_i where i is the smallest allowable index. The strategy f' now simulates what f would do from s_i. Two cases: (1) if f moves to a vertex a_j on C then f' moves to a, or (2) if f moves to a vertex a' not on C then also f' moves to a'. In the first instance the strategy f' forces a play that toggles between a and s in B' until case (2) holds. (And this must happen since f is an update strategy.)

Case $v_n' \neq s$. Here the strategy is simply $f'(v_0', \ldots, v_n') = p(f(v_0, \ldots, v_n))$. That is, the play follows the strategy f on the simulated game history of B.

The strategy f' for the Survivor is an update strategy since p is a mapping from B onto B' (i.e., if all vertices of B are infinitely repeated via f then all vertices of B' are infinitely repeated via f'). □

With respect to the above proof, Figure 3 shows how a forced cycle of B is reduced to a smaller forced cycle (of length 2) in B'.

Theorem 11. There exists an algorithm that decides whether a bipartite update game B is a bipartite update network in time $O(n \cdot m)$, where n and m are the order and size of the underlying graph.

Proof. We show that finding a cycle that is guaranteed to exist by Lemma 9 takes time at most $O(m)$ and that producing B' from B in Lemma 10 takes time at most $O(n+m)$. Since we need to recursively do this at most n times the overall running time is shown to be $O(n \cdot m)$.

The algorithm terminates whenever a forced cycle of length at least four is not found. It decides whether the current bipartite graph is a update network by simply checking that $S = \{s\}$ and out-degree$(s) = |A|$. That is, the singleton Survivor vertex is connected to all Adversary vertices.

Let us analyze the running time for finding a forced cycle C. Recall the algorithm begins at any vertex s_1 and finds an in-neighbor a_1 (of s_1) of out-degree 1 with $(s_2, a_1) \in E$ where $s_2 \neq s_1$. This takes time proportional to the number of edges incident to s_1 to find such a vertex a_1. Repeating with s_2 we find an a_2 in time proportional to the number of edges into s_2, etc. We keep a boolean array to indicate which s_i are in the partially constructed forced path (i.e., the look-up time will be constant time to detect a forced cycle of length at least 4). The total number of steps to find the cycle is at most a constant factor time the number of edges in the graph.

Finally, we can observe that building B' from B and C of Lemma 10 runs in linear time by the definition of S', A' and E'. Note that if the data structure for graphs is taken to be adjacency lists then E' is constructed by copying the lists of E and replacing one or more vertices s_i's or a_j's with one s or a, respectively. □

The above result indicates the structure of bipartite update networks. These are basically connected forced cycles, with possibly other legal moves for some of the Survivor and the Adversary vertices. Figure 4 shows a constructed example of one such bipartite update network. The Survivor's strategy is to systematically repeat the forced cycles and 'detour' to cover the remaining non-forced Adversary vertices on a periodic basis.

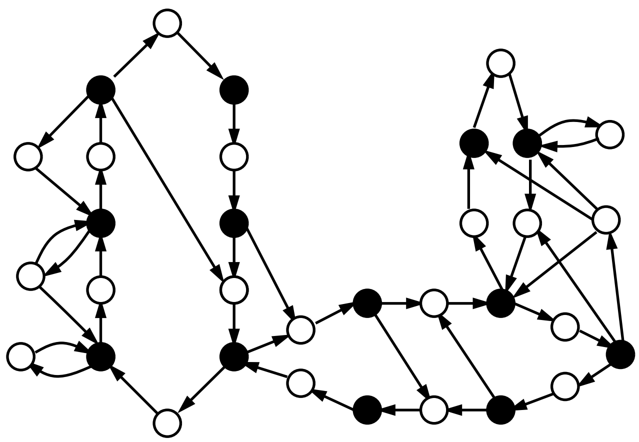


Figure 4: Illustrating the structure of bipartite update networks with Survivor vertices (black) and Adversary vertices (white).

4 Recognizing Update Networks

We now want to present an algorithm to decide whether a given update game is also an update network. Our idea is to take an update game $\mathcal{G}$ and implicitly transform it into a bipartite game $\mathcal{B}_G$. (Note $\mathcal{B}_G$ will *not* be a bipartite update game, as described in Section 3.) We then show how to decide if G is an update network by checking if the Survivor has a winning strategy for every initial configuration of $\mathcal{B}_G$. Recall that in a bipartite game the Adversary and the Survivor only move from one of the vertex partitions of the graph.

We define the game $\mathcal{B}_G = (B, W)$ from an update game $\mathcal{G} = (G, \{V(G)\})$ as follows:

$$\begin{array}{rcl} V(B) & = & \{(v_S \mid v \in V(G)\} \cup \{v_A \mid v \in V(G)\} \\ E(B) & = & \{(v_S, u_A) \mid (v,u) \in E(G)\} \cup \{(v_A, u_S) \mid (v,u) \in E(G)\} \\ W & = & \{Y \mid \forall v \in V(G), \exists w \in Y (w = v_S \text{ or } w = v_A)\} \end{array}$$

Note that the graph B is only twice the size of G but the explicit storage for the winning sets W is exponential in the size of $\mathcal{G}$'s winning sets $\{V(G)\}$. Figure 5 shows a small example of the construction of B from G.

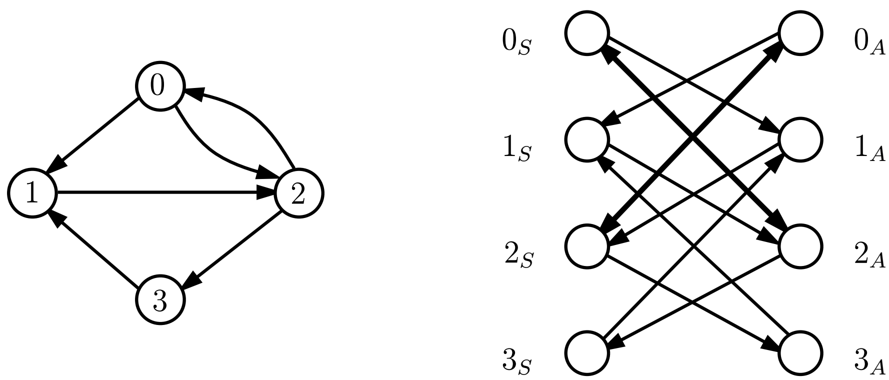


Figure 5: Mapping an update game (graph G) to a bipartite game (graph B).

The vertices of B will correspond to a vertex/player combination of the game $\mathcal{G}$. We have the following equivalence.

Lemma 12. The game $\mathcal{G}$ is an update network if and only if the Survivor has a winning strategy for every initial configuration of $\mathcal{B}_G$.

Proof. First assume $\mathcal{G}$ is an update network. For any initial configuration (v, X) of the game $\mathcal{G}$ the Survivor has a winning strategy f. The Survivor can use this strategy f for the initial configuration v_X in the game $\mathcal{B}_G$. Since f forces all vertices of G to be visited infinitely often, at least one of the v_A or v_S is visited infinitely often in B (for all $v \in G$).

Now assume that the Survivor has a winning strategy f' for $\mathcal{B}_G$ starting at vertex v_X. Every persistent set of vertices Y that occur when the Survivor uses f' is in W. The Survivor can simulate f' (on $\mathcal{B}_G$) for the game $\mathcal{G}$ with initial configuration (v, X) and win the game. $\square$

We now define for any subset of vertices V' of a bipartite game G the *closure* Forced$^*(V')$. This is the set of vertices (containing V') that the Survivor has a strategy to force the Adversary to visit at least one vertex of V'. We have the following algorithm to compute Forced$^*(V')$.

algorithm FindForced($V' \subseteq V(G)$) for bipartite graph $G = (S \cup A, E)$

```
1   Queue NewVerts = V'
    Set F = V'
2   while Vertex v in NewVerts.head() do
        NewVerts.remove(v)
3       if v ∈ A then
            Set F' = inNeighbors(v)
            NewVerts.append(F')
            F = F ∪ F'
        endif
4       if v ∈ S then
            Set F' = ∅
5           for Vertex u in inNeighbors(v) do
                if outNeighbors(u) ⊆ F then  F' = F' ∪ {u}
            endfor
            NewVerts.append(F')
            F = F ∪ F'
        endif
    endwhile
    return F
    end
```

We prove the correctness of this algorithm below.

Lemma 13. Algorithm FindForced computes $\text{Forced}^*(V')$ for a bipartite graph $G = (S \cup A, E)$.

Proof. We show that for every vertex v, v is in $\text{Forced}^*(V')$ if and only if v is returned in F by the algorithm FindForced. To do this we assign a number, called *rank*, to each vertex of the graph. The rank indicates the number of forced moves needed to reach V' from a vertex. The *rank* function is recursively defined as follows:

$$rank(v) = \begin{cases} 0 & \text{if } v \in V' \\ \min_{u \in \text{outNeighbors}(v)} rank(u) + 1 & \text{if } v \in S \text{ and } v \notin V' \\ \max_{u \in \text{outNeighbors}(v)} rank(u) + 1 & \text{if } v \in A \text{ and } v \notin V' \\ \infty & \text{otherwise} \end{cases}$$

We now show that $v \in \text{Forced}^*(V')$ if and only if $rank(v) < \infty$. Suppose $rank(v) = n < \infty$. If $n = 0$ then $v \in V'$. Otherwise consider two cases. If $v \in S$ then v is in the closure since at least one neighbor u of v has smaller rank (i.e., the Survivor can move to u and $rank(v) \leq rank(u) + 1$). If $v \in A$ then v is in the closure since all neighbors of v have rank less n (i.e., the any move of the Adversary moves to a vertex to u of rank less than n). Now suppose $rank(v) = \infty$ and again consider two cases. If $v \in S$ then all neighbors of v have rank equal to ∞ (i.e., the Survivor can not reach V' from v). If $v \in A$ then there is at least one neighbor u of v with rank equal to ∞ (i.e., the Adversary can move to u that is not in the closure.)

One can see that the algorithm FindForced adds a vertex v to F if and only if it has finite rank. The algorithm implicitly labels a vertex v of $S \cup A$ by the iteration count of the while loop at line 2 when v is added to F (the vertices V' are labeled with count 0). Hence if a vertex is labeled then it has finite rank. Statement 3 of the algorithm corresponds to the case $v \in S$ and $v \notin V'$ of the definition of rank while Statements 4–5 correspond to the case $v \in A$ and $v \notin V'$. This means that if v has finite rank then it will be labeled by the algorithm. □

Lemma 14. For bipartite games, there exists an algorithm that runs in time $O(m)$, where m is the size of the graph, that computes $\text{Forced}^*(V')$.

Proof. We show how to modify the algorithm FindForced to run in $O(m)$ time. The algorithm as listed needs to process each vertex in the queue NewVerts at most once and for each of these vertices access its in-neighbors. So excluding the loop at line 5 the algorithm runs in $O(m)$ steps. The process time, as listed, to check whether $\text{outNeighbors}(u) \subseteq F$ takes at most $O(n)$ time. Hence, FindForced runs in time $O(n{\cdot}m)$.

We now explain how to reduce the running time of the loop at line 5 of algorithm FindForced to constant time. Instead of checking the set membership $\text{outNeighbors}(u) \subseteq F$ we do the following. We keep an array of integers Deg that indicates for each vertex how many neighbors are not currently in F. The entry for vertex x is initially defined as the out-degree of x. Whenever a vertex y is added to F we decrement the entry for each in-neighbor z of y by one. We can now replace the condition $\text{outNeighbors}(u) \subseteq F$ by testing whether $Deg[u] = 0$, which can be done in constant time. □

Recall Lemma 12 states that a game $\mathcal{G}$ is an update network if and only if the Survivor has a winning strategy for every initial configuration of $\mathcal{B}_G$. The next Theorem also characterizes update networks (not necessarily bipartite games) by using the closure operator.

Theorem 15. A game $\mathcal{G} = (G, \{V(G)\})$ is an update network if and only if, for all $v \in V(G)$, $\text{Forced}^*(\{v_S, v_A\}) = V(B)$ in the corresponding bipartite game $\mathcal{B}_G = (B, W)$.

Proof. Suppose for an update network exists over graph G such that $\text{Forced}^*(\{v_S, v_A\}) \neq V(B)$ for some $v \in V(G)$. Take any vertex x of B that does not belong to this closure. Using the proof of Lemma 13 we see that the Survivor can not force the play to visit v_S or v_A from vertex $x \in V(B)$. Thus the Adversary wins game $\mathcal{B}_G$ beginning from x. By Lemma 12 the graph G can not be an update network.

We now prove the other implication of the theorem. It suffices to show that the Survivor can win the game $\mathcal{B}_G$ from any starting vertex. We use the fact that $\text{Forced}^*(\{v_S, v_A\}) = V(B)$, for all $v \in V(G)$, to build a winning strategy for the Survivor in $\mathcal{B}_G$. Order the vertices of G as $v^1, v^2, \ldots, v^n$. Let x be a starting vertex of B. The Survivor can use algorithm FindForced to visit either v^1_S or v^1_A. Next the Survivor can force the play to visit to either v^2_S or v^2_A, then either v^3_S or v^3_A, etc. The Survivor then repeats the forced plays between the pairs (v^i_S or v^i_A) and ($v^{i+1 \bmod n}_S$ or $v^{i+1 \bmod n}_A$) which yields a winning set of W. □

Using the previous lemma and theorem we can efficiently recognize update networks.

Theorem 16. There exists an algorithm that decides whether a update game $\mathcal{G}$ is an update network in time $O(n \cdot m)$, where n and m are the order and size of the underlying graph.

Proof. We can construct the bipartite graph B from the game $\mathcal{B}_G$, which corresponds to $\mathcal{G} = (G, \{V(G)\})$, in linear time. We then invoke Lemma 14 for each pair of vertices $\{v_A, v_S\}$ for $v \in V(G)$. By using Theorem 15, we accept input if Forced$^*(\{v_S, v_A\}) = V(B)$, for all $v \in V(G)$. The total running time is $n = |V(G)|$ multiplied by the time needed to compute the closure (of two vertices v_A and v_S) in B. This product is $O(n \cdot m)$. □

5 Conclusion

In this paper we have presented a game-theoretic model of infinite duration processes. A particular emphasis is given to a class of networks whose objective is to continuously update all the nodes with consistent data. We have shown that it is algorithmically feasible to recognize update networks. That is, we have provided an algorithm which solves the update game problem in $O(n \cdot m)$ time. Moreover, our algorithm for the case of bipartite update games can be used to give a characterization of bipartite update networks.

There are many open questions that still need to be investigated in this area. For example, one can try to characterize those update games for which the update network problem is decidable in linear time. One can also study the question of finding feasible algorithms for games whose winning conditions are more complex than the one for update games. For the latter case, we want to efficiently extract winning strategies (if they exist for the Survivor) for each set of vertices in the winning set of a game.

The games considered in this paper occur over finite graphs. These games can be generalized to games over different finite models (such as hypergraphs). We would like to know which of these generalized game problems are tractable.

References

[1] R. McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic* 65 (1993), 149–184.

[2] A. Nerode, J. Remmel and A. Yakhnis. McNaughton games and extracting strategies for concurrent programs. *Annals of Pure and Applied Logic* 78 (1996), no. 1-3, 203–242.

[3] R.J. Büchi and L.H Landweber. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.* 138 (1969), 295–311.

[4] T.J. Schäfer. Complexity of some two-person perfect-information games. *J. Computer & System Sciences* 16 (1978), 185–225.