Update Games and Update Networks

MICHAEL J. DINNEEN and BAKHADYR KHOUSSAINOV,
Department of Computer Science, University of Auckland, Auckland,
New Zealand.
E-mail: {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.

Keywords: Infinite games, update networks.

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.).

Over time, each system 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 a 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 the opposite.

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 (or arcs) correspond to the legal state changes, called moves (or transitions), of the system.

DEFINITION 1.1
An infinite duration game $\mathcal{G}$ is a finite (directed) graph $G = (V, E)$, a family W of subsets of V, and two players (the Survivor and the Adversary). We require that each vertex of G has out-degree of at least one. 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

 © King's College Publications

$v \in V$.

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 an infinite 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 some history of the play. Note that either the Survivor or the Adversary may begin the play. The Survivor wins a play if the set of persistent vertices of the play is a winning set, 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.

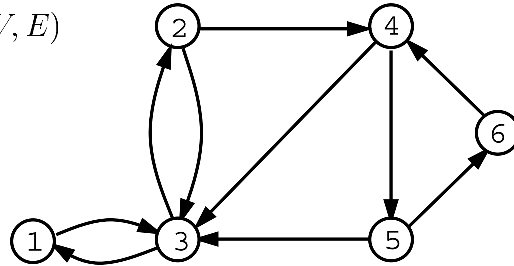
EXAMPLE 1.2
In Figure 1 we present a game $\mathcal{G} = (G, W)$. As an example of a winning strategy for the Survivor consider the initial configuration $(4, \text{Adversary})$. If the Adversary moves to vertex 3 then the Survivor simply moves to vertex 1 and the game repeats between those two vertices (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 loses no matter what choice is made at vertex 5.

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]. Our work focuses on a subclass of the games considered by these authors. Nerode et al. provide an algorithm for deciding McNaughton games. Their algorithm runs in exponential time of the graph size for certain inputs. For our games we provide two simple polynomial time algorithms for deciding update networks, partially based on the structural properties of the underlying graphs. We also note that several earlier papers have dealt with finite duration games on automata and graphs (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

$G = (V, E)$



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

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

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

FIG. 1. Example of an infinite duration game.

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 2.1
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 2.2
The graph displayed below in Figure 2 is an update network. Notice that all cycles are of odd length so that the Survivor and the Adversary alternately control the vertices with more than one possible move. The Survivor can use its opportunities to visit all vertices of the graph.

3 Bipartite Update Networks

We first study a specific class of update games on bipartite graphs, called bipartite update games. For these games 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 every edge is 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 definition, we assume that the Survivor moves from S and the Adversary

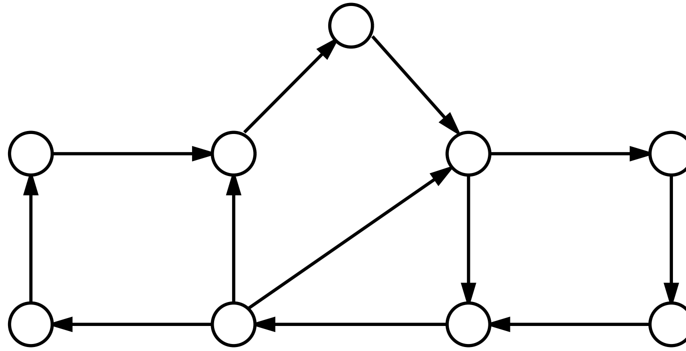


FIG. 2. A simple example of an update game which is an update network.

moves from A. In essence the vertices (in these bipartite games) are owned by the two players of the game. Thus, for these games, there are only $|V|$ game configurations where each vertex v determines a unique configuration depending on whether v is in S or A. In the next section we return to the update games defined in Definition 2.1.

DEFINITION 3.1
A bipartite update network is a bipartite graph $(V = A \cup S, E)$ of a bipartite 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.)

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 3.2
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 $a \in A_s$ then since out-degree$(a) > 1$, the Adversary moves to s' (of S), 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. Note, by the previous lemma, this set will be non-empty for games played on bipartite update networks.

LEMMA 3.3
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 $a \in$ Forced(s), such that (s', a, s) is a directed path.

PROOF. Take any $s \in S$ and consider $F =$ Forced(s). By the Lemma 3.2 F is not empty. If there is an $s' \neq s$ adjacent to a vertex (i.e., an in-neighbor) in F we are done. Otherwise, all $s' \neq s$ are not adjacent to any vertex in F. Thus there exists an a' not in F from which the Adversary has a winning strategy by not moving to s. This contradicts B being a bipartite update network. ∎

DEFINITION 3.4
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 $a_i \in$ Forced(s_i) and $s_i \in S$. (Note that forced cycles have even length since the graph is bipartite.)

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

LEMMA 3.5
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 3.3 there exists a path (s_2, a_1, s_1) in B such that $s_2 \neq s_1$ and $a_1 \in$ 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 $a_2 \in$ Forced(s_2). If $s_3 = s_1$ we are done. Otherwise repeat Lemma 3.3 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. ∎

Thus, 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. We now present a contraction method that helps us decide if a bipartite game is a bipartite update network.

Let $B = (S \cup A, E)$ be a bipartite update game with a forced cycle $C = (a_k, s_k, \ldots, a_2, s_2, a_1, s_1)$ of length at least 4. We can define a contracted bipartite update game $B' = (S' \cup A', E')$ as follows. For new vertices a and s let

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

With E'' being the induced edges of the subgraph $B \setminus \{s_1, a_1, \ldots, s_k, a_k\}$ let

$$\begin{aligned} E' = &\{(s, a') \mid a' \in A' \text{ and } (s_i, a') \in E, \text{ for some } i \leq k\} \cup \\ &\{(a', s) \mid a' \in A' \text{ and } (a', s_i) \in E, \text{ for some } i \leq k\} \cup \\ &\{(s', a) \mid s' \in S' \text{ and } (s', a_i) \in E, \text{ for some } i \leq k\} \cup \{(a, s), (s, a)\} \cup E''. \end{aligned}$$

The next lemma shows the relationship between game B and the reduced game B'.

LEMMA 3.6
If $B = (S \cup A, E)$ is a bipartite update game with a forced cycle C of length at least 4 then the contracted bipartite update game $B' = (S' \cup A', E')$ is a bipartite update network if and only if B is one.

PROOF. We 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

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

Then any play history of B is mapped, via the function $p(v) = v'$, onto a play history of B'. Consider a play history $v_0, v_1, \ldots, v_n$ of B that starts at vertex v_0 and $v_n \in S$. Let f' be a winning strategy in game B' 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 considering 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 $f'(v'_0, \ldots, v'_n) = a'$ where $a' \neq a$ 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 $f'(v'_0, \ldots, v'_n) = a$ and the play is extended by picking an $a_k \in C$ such that $(v_n, a_k) \in E$.
Case $v'_n \neq s$. If $f'(v'_0, \ldots, v'_n) = a' \neq a$ 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 $v'_0 = s$. 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 $v'_k = a' \neq a$ of the play history that is in A' and $v'_{k+1} = s$. Thus in the game on B, the Adversary from a' elects to move to some s_i in C. Since in game B' the Adversary may have fewer choices, we can pick, without loss of generality, that it moved to s_i where i is the smallest allowable index. The choice of any index i is validated because the cycle C is a forced cycle. 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'). ∎

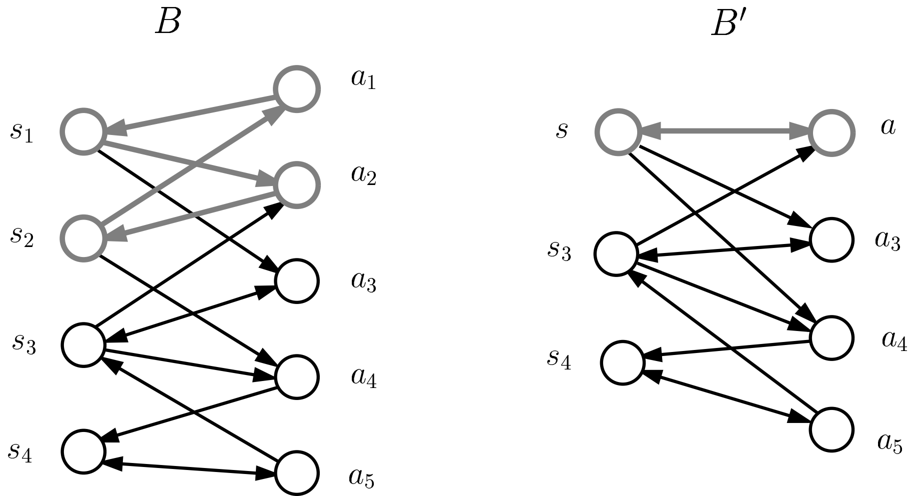


FIG. 3. Showing the bipartite update game reduction of Lemma 3.6.

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

For the next result let n denote the order (number of vertices) and m denote the size (number of edges) of a graph.

THEOREM 3.7
There exists an algorithm that decides whether a bipartite update game B is a bipartite update network in time $O(n \cdot m)$.

PROOF. We show that finding a cycle that is guaranteed to exist by Lemma 3.5 takes time at most $O(m)$ and that producing B' from B in Lemma 3.6 takes time at most $O(n+m)$. We can also detect in time at most $O(m)$ when a forced cycle of length at least 4 does not exist. 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 implied by Lemma 3.5 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 times the number of edges in the graph.

Finally, we can observe that building the contracted bipartite game B' from B and C 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.

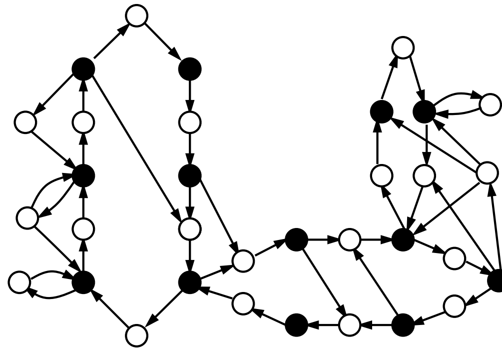


FIG. 4: Illustrating the structure of bipartite update networks with Survivor vertices (black) and Adversary vertices (white).

We list all the non-isomorphic bipartite networks of order at most 5 in Figure 5. Note that 18 of the 19 networks of order 5 were generated from three of the networks of order 4 (see those displayed on the top row). This was done by systematically adding a new adversary node with all possible combinations in-degrees and out-degrees of 1 and 2. Note that the first four graphs in the first column and the first and fourth graphs in the last column are minimal in the sense that all edges are essential for these graphs to be bipartite update networks.

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 the graph G (of $\mathcal{G}$) 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

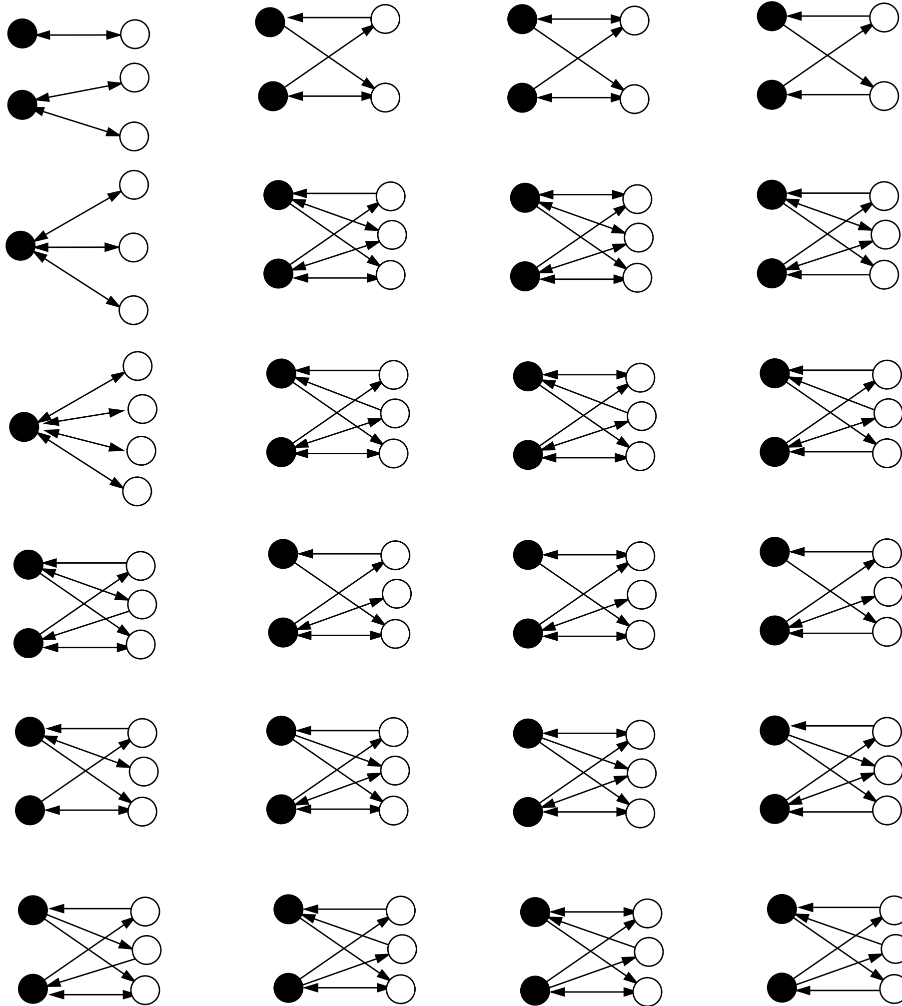


FIG. 5: All small bipartite update networks with Survivor vertices (black) and Adversary vertices (white).

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:

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

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 6 shows a small example of the construction of B from G.

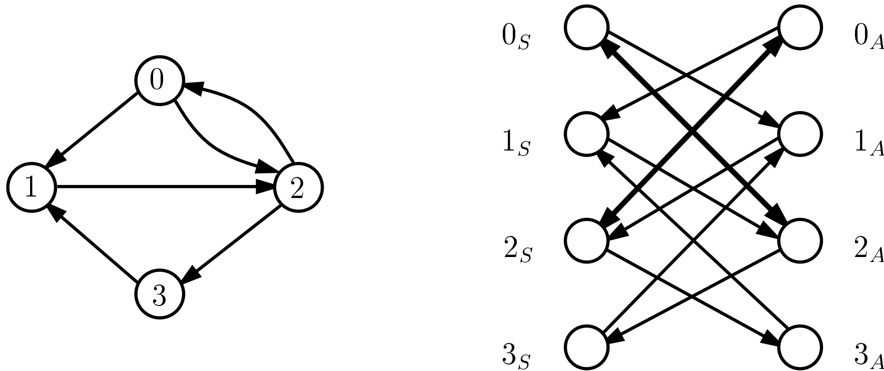


FIG. 6. Mapping an update game (graph G) to a bipartite game (graph B).

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

LEMMA 4.1
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 $\mathcal{B}_G$. (Recall that in the bipartite game $\mathcal{B}_G$ the Survivor can only start from a vertex v_S). 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. ∎

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' ⊆ V(G)) for bipartite graph G = (S ∪ 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 ∪ 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 ∪ F'
        endif
    endwhile
    return F
    end
```

We prove the correctness of this algorithm below.

LEMMA 4.2
Algorithm FindForced computes Forced*(V') for a bipartite graph $G = (S \cup A, E)$.

PROOF. We show that for every vertex v, v is in Forced*(V') if and only if v is returned in F by the algorithm FindForced. To do this we assign a number, called the 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 inductively defined as follows:

1. If $v \in V'$ then $rank(v) = 0$.
2. Case $v \in S$ and $rank(v)$ is not defined. Assume all out-neighbors of v of rank of at most i have been defined. Then $rank(v) = i + 1$ if there exists an $u \in$ outNeighbors(v) with $rank(u) = i$.
3. Case $v \in A$ and $rank(v)$ is not defined. Assume all out-neighbors of v have defined rank. Then $rank(v) = i + 1$ if each $u \in$ outNeighbors(v) has $rank(u) \leq i$.

If $v \in G$ does not get ranked in the above process then we set $rank(v) = \infty$. We now show that $v \in$ 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 $rank(u) < rank(v)$). If $v \in A$ then v is in the closure since all neighbors of v have rank less than n (i.e., any move of the Adversary moves to

a vertex u of rank less than n). Suppose $rank(v) = \infty$. We want to show that the Adversary has a strategy that does not allow the Survivor to reach V'. We note that following two observations. If $v \in S$ then all neighbors of v have rank equal to ∞ by definition of the rank function (i.e., the Survivor can not reach V' from v). Also by definition, 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). Now the Adversary's strategy to avoid V' is the following. For any $v \in A$ with $rank(v) = \infty$ move to a neighbor vertex u such that $rank(u) = \infty$. Clearly this strategy causes all plays to stay on a subset of the set $V'' = \{w \mid rank(w) = \infty\}$ and $V'' \cap V' = \emptyset$.

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 4.3
For bipartite games, there exists an algorithm that runs in time $O(m)$, where m is the size of the graph, that computes 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 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 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 outNeighbors$(u) \subseteq F$ by testing whether $Deg[u] = 0$, which can be done in constant time. ∎

Recall Lemma 4.1 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 4.4
A game $\mathcal{G} = (G, \{V(G)\})$ is an update network if and only if for all $v \in V(G)$ Forced*$(\{v_S, v_A\}) = V(B)$ in the corresponding bipartite game $\mathcal{B}_G = (B, W)$.

PROOF. Suppose there is an update network, with graph G, such that 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 4.2 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 4.1 the graph G (of the game $\mathcal{G}$) can not be an update network.

We now prove the other implication of the theorem. By Lemma 4.1, it suffices to show that the Survivor can win the game $\mathcal{B}_G$ from any starting vertex. We use the fact that 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_S^1 or v_A^1. Next the Survivor can force the play to visit to either v_S^2 or v_A^2, then either v_S^3 or v_A^3, etc. The Survivor then repeats the forced plays between the pairs (v_S^i or v_A^i) and ($v_S^{i+1 \bmod n}$ or $v_A^{i+1 \bmod n}$) which yields a winning set of W. ∎

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

THEOREM 4.5
There exists an algorithm that decides whether an 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 with respect to the size of G. We then invoke Lemma 4.3 for each pair of vertices $\{v_A, v_S\}$ for $v \in V(G)$. By using Theorem 4.4, we accept the 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.

Acknowledgment

We thank the two anonymous referees who provided detailed and useful comments on an earlier version of this paper.

References

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

[2] A. Nerode, J. Remmel and A. Yakhnis. McNaughton games and extracting strategies for concurrent programs. Annals of Pure and Applied Logic, 78, 203–242, 1996.

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

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

Received 1 November 1999.