Electronic Colloquium on Computational Complexity, Report No. 12 (2006)

Efficient Computation of Nash Equilibria for Very Sparse Win-Lose Bimatrix Games

Bruno Codenotti[1] Mauro Leoncini[2] Giovanni Resta[1]

Abstract

It is known that finding a Nash equilibrium for win-lose bimatrix games, i.e., two-player games where the players' payoffs are zero and one, is complete for the class $PPAD$.

We describe a linear time algorithm which computes a Nash equilibrium for win-lose bimatrix games where the number of winning positions per strategy of each of the players is at most two.

The algorithm acts on the directed graph that represents the zero-one pattern of the payoff matrices describing the game. It is based upon the efficient detection of certain subgraphs which enable us to determine the support of a Nash equilibrium.

1 Introduction

In 1951 Nash proved that any n-player game has an equilibrium in the mixed strategies [9]. The proof was based on a fixed point argument, and left open the associated computational question of finding such an equilibrium. In 1964, Lemke and Howson introduced an algorithm for the computation of a Nash equilibrium in 2-player games. In the worst case, this algorithm has an exponential running time [11]. It provides us with another, different from Nash's, proof of the existence of an equilibrium.

In 1994 Papadimitriou introduced a complexity class, $PPAD$, which captures a wealth of equilibrium problems, e.g., the market equilibrium problem as well as Nash equilibria for n-player games [10]. Problems complete for this class include a (suitably defined) computational version of the Brouwer Fixed Point Theorem.

In 2005 a flurry of results appeared, where first the $PPAD$-completeness of 4-player games [5], then of 3-player games [2, 6], and finally of 2-player games [3] were proven. In particular, the latter hardness result by Chen and Deng came as a sort of surprise, since the 2-player case was conjectured to be computationally tractable. Combined with a result by Abbott, Kane, and Valiant [1], it also implies the $PPAD$-completeness of win-lose bimatrix games, i.e., 2-player games where the players' payoffs are zero and one.

Therefore it seems unlikely that polynomial time algorithms exist for win-lose bimatrix games. This fact makes it worthwhile to analyze restricted versions of the problem which might be endowed with computationally useful structural properties.

In this paper, we consider some restricted win-lose games, where the zero-one matrices describing the payoffs of the players have at most two ones per row and column. Following [4],

[1]IIT-CNR, Via Moruzzi 1, Pisa (Italy). E-mail: [bruno.codenotti, giovanni.resta]@iit.cnr.it.

[2]Dipartimento di Ingegneria dell'Informazione. Università di Modena e Reggio Emilia, Modena (Italy) . E-mail: leoncini@unimo.it.

ISSN 1433-8092

we cast the problem of computing an equilibrium for win-lose games in terms of finding a *good assignment* in a directed graph - see Section 2 below for proper definitions.

The restriction on the zero-one pattern induces very sparse directed graphs. We show how to efficiently detect suitable subgraphs of these sparse graphs, which lead to the discovery of the support of a Nash equilibrium, and to the actual determination of the equilibrium strategies.

This paper is organized as follows. In Section 2 we introduce the game theoretic notions to be used in the paper. After defining the concept of Nash equilibrium for 2-player games in normal form, we show how the computation of a Nash equilibrium for a win-lose 2-player game is equivalent to the computation of a *good assignment* in a directed graph. In Section 3 we informally describe our algorithm, with the help of illustrations and examples. In Section 4 we give the pseudocode of the algorithm, formally prove its correctness, and analyze its running time. Finally in Section 5 we present some conclusions and open questions.

2 Background

We consider 2-player games in *normal form*. These games are described by a pair (A, B) of matrices, whose entries are the *payoffs* of the two players, called row and column player. $A = (a_{ij})$ is the payoff matrix of the row player, and $B = (b_{ij})$ is the payoff matrix of the column player.

The rows (resp. columns) of A and B are indexed by the row (resp. column) player's *pure strategies*.

The entry a_{ij} is the payoff to the row player, when she plays her i-th pure strategy and the opponent plays his j-th pure strategy. Similarly, b_{ij} is the payoff to the column player, when he plays his j-th pure strategy and the opponent plays her i-th pure strategy.

A *mixed strategy* is a probability distribution over the set of pure strategies which indicates how likely it is that each pure strategy is played. More precisely, in a mixed strategy a player associates to her i-th pure strategy a quantity p_i between 0 and 1, such that $\sum_i p_i = 1$, where the sum ranges over all pure strategies.

Let us consider the game (A, B), where A and B are $m \times n$ matrices. In such a game the row player has m pure strategies, while the column player has n pure strategies. Let x (resp. y) be a mixed strategy of the row (resp. column) player. Strategy x is the m-tuple $x = (x_1, x_2, \ldots, x_m)$, where $x_i \geq 0$, and $\sum_{i=1}^{m} x_i = 1$. Similarly, $y = (y_1, y_2, \ldots, y_n)$, where $y_i \geq 0$, and $\sum_{i=1}^{n} y_i = 1$.

When the pair of mixed strategies x and y is played, the entry a_{ij} contributes to the expected payoff of the row player with weight $x_i y_j$. The expected payoff of the row player can be evaluated by adding up all the entries of A weighted by the corresponding entries of x and y, i.e., $\sum_{ij} x_i y_j a_{ij}$. This can be rewritten as $\sum_i x_i \sum_j a_{ij} y_j$, which can be expressed in matrix terms as[1] $x^T A y$. Similarly, the expected payoff of the column player is $x^T B y$.

Definition 1 [Nash Equilibrium] A pair of mixed strategies (x, y) is in Nash equilibrium if $x^T A y \geq x'^T A y$, for all stochastic m-vectors x', and $x^T B y \geq x^T B y'$, for all stochastic n-vectors y'.

We say that x (resp. y) is a *Nash equilibrium strategy* for the row (resp. column) player.

The set of indices such that $x_i > 0$ (resp. $y_i > 0$) is called the *support* of the Nash equilibrium strategy x (resp. y).

[1] We use the notation x^T to denote the transpose of vector x.

The following Lemma shows that it is possible to restrict our attention to bimatrix games where one of the players' payoff matrix is the identity.

Lemma 2 ([4]) *Let A, B be two $m \times n$ matrices with nonnegative entries, where A (resp., B) has at least one nonzero entry in each row (resp., column). Let $C = \begin{pmatrix} 0 & B \\ A^T & 0 \end{pmatrix}$ and let I be the $(m+n) \times (m+n)$ identity matrix. The Nash equilibria of the game (A, B) are in one-to-one correspondence with the Nash equilibrium strategies of the row player in the game (I, C).*

We now consider games of the form (I, C), where C is a square matrix with zero-one entries.

To avoid trivial pure strategy Nash equilibria, we assume that the matrices I and C do not have entries equal to 1 in the same position, i.e., we assume that the entries on the main diagonal of C are all zero.

Following [4], we now define the notion of *good assignment* in a directed graph G, and then show (Lemma 4 below) that it is equivalent to the notion of Nash equilibrium for the game (I, A), where A is the adjacency matrix of G.

Definition 3 [Good Assignment]

Let $G = (V, E)$ be a directed graph. Let x be an assignment of nonnegative weights to the vertices of G. We can assume that x is normalized, i. e. $\sum_i x_i = 1$. The *income* $i_x(v)$ of a vertex v is the sum of weights of vertices u which point to v, i. e. $i_x(v) = \sum_{u:(u,v)\in E} x_u$. A vertex v is *happy* if it has highest income (i. e. $i_x(v) \geq i_x(u)$ for all $u \in V$). A vertex v is *working* if it has nonzero weight (i. e. $x(v) > 0$). An assignment x is *good* if all the working vertices are happy.

Lemma 4 ([4]) *Let (I, A) be a bimatrix game, where A is a zero-one matrix with zero entries along the main diagonal. Let $G[A]$ be the digraph with adjacency matrix A. The Nash equilibrium strategies of the row player in (I, A) are in one-to-one correspondence with the good assignments in $G[A]$.*

The proof of Lemma 4 is quite simple. For the sake of self-containment, we now sketch the idea. Consider a good assignment x for $G[A]$. It has the property that the entries of the row vector $x^T A$ are maximal for indices j such that $x_j > 0$. If needed, we can scale the entries of x so that $\sum_i x_i = 1$. Let k be the size of the support of x, i.e., k is the number of positive entries of x. Consider the vector y such that $y_j = \frac{1}{k}$, if $x_j > 0$, and $y_j = 0$, if $x_j = 0$. It is easy to check that the pair (x, y) is a Nash equilibrium for the game (I, A).

Based on Lemmas 2 and 4, we can focus on the computation of a good assignment.

3 An informal description of the algorithm

In this section we provide an informal description of the algorithm for the computation of a good assignment. In the next section, we will then give a more formal presentation, with proper definitions, the pseudocode, the proof of correctness, and the analysis of the running time.

First of all, notice that we can restrict ourselves to finding a good assignment in a strongly connected digraph. In fact, if the graph is not connected, we can find a good assignment in one of its connected components and assign zero weight to all vertices in the other components. If the graph is connected, but not strongly connected, then its vertices can be partitioned into two subsets V' and V'' such that the graph induced by the vertices in V'' is strongly connected, and

the arcs joining vertices from V' and vertices from V'' are all directed towards the vertices in V''. Therefore a good assignment for the original graph can be obtained via an extension of a good assignment for V'', obtained by setting to zero the weights of the vertices in V'.

The reduction outlined above can be computed by standard algorithms in time linear in the number of vertices plus the number of edges of the graph.

The algorithm is composed of two main phases. The first one partitions the vertices of the input graph into two subsets, where the vertices of one subset can be assigned weight zero. The input to the second phase of the algorithm is the subgraph induced by the second subset of vertices. The output of this phase is a further smaller set of vertices for which it is easy to assign nonzero weights that form the support of the good assignment for the original graph.

Our approach thus consists of the decomposition of a strongly connected digraph G in two components, the first one with a structure that admits an easy to compute good assignment with positive weights, and the other one composed by vertices whose weights can be set to zero.

This idea is illustrated by the example shown in Fig.1 (left). The vertices belonging to the subgraph G' (a cycle) are assigned weight 1, while all the other vertices are assigned weight 0. All vertices with weight 1 have income 1 and all the vertices with weight 0 have income *at most* 1.

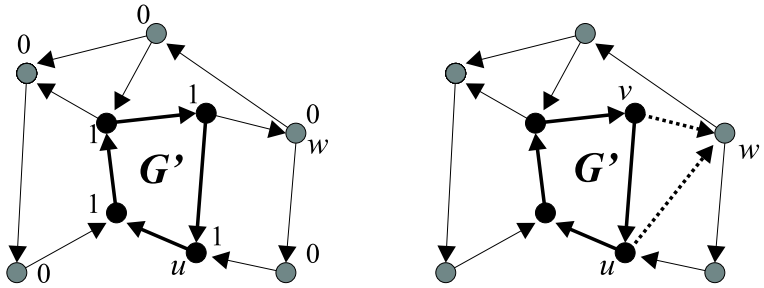

Figure 1: On the left a valid solution, obtained by equating to one the weights of the vertices belonging to the subgraph G'. On the right, the addition of the edge (u, w) creates a "bump", disrupting the previous solution.

A major obstacle in the search for such a decomposition is the potential presence of two arcs exiting from G' and pointing to the same vertex (w in Fig.1(right)) not in G'. Indeed, if we add the arc (u, w) to G, then, unless we change the weights of the nodes in G', the node w will have income 2, which prevents the current assignment from being a good assignment.

We will call a *"bump"* a configuration given by two arcs exiting a subgraph G' and pointing to the same vertex, not belonging to G' (see Fig.2).

The goal of the first phase of our algorithm will be that of finding a subgraph G' of G with two desirable properties:

- G' can be represented as a directed cycle C, containing all the vertices of G', plus a number of *chords*, i.e., arcs of G' not in C.
- G' has no bumps in G, i.e., there are no vertices $u, v \in G'$ and $w \in G \backslash G'$ such that both edges (u, w) and (u, w) exist in G.

Note that, contrary to what was the case in the example of Fig.1, a subgraph G' satisfying the two properties above does not yet allow us to determine a trivial assignment of weights,

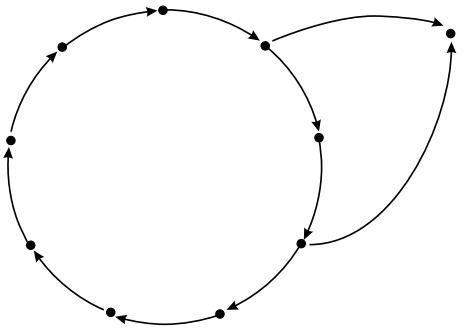

Figure 2: A cycle with a bump.

because of the presence of chords. In the second phase of the algorithm we will take care of the chords, by finding a suitable subgraph of G' with additional properties.

We now show how to detect such a subgraph G'. Given a graph G, in order to obtain a cycle $C \subseteq G$ we can start from an arbitrary vertex u in G, and then just follow one of its outgoing arcs (since G is strongly connected, there is at least one such arc). Repeating this procedure we will eventually reach an already visited vertex, thus obtaining a cycle.

However there is no guarantee that such a cycle C will be bump-free, i.e., it might contain two vertices u and v, and a third vertex $w \in G \backslash C$, where the edges $u \to w \leftarrow v$ belong to G. The easiest way to make sure that such a bump does not occur during the construction of the cycle C is to follow the arc $u \to w$ if we are visiting u, and the arc $v \to w$ if we are visiting v. Either way w will be included in the final cycle C if any of u and w is included in C, thus preventing the adjacency configuration $u \to w \leftarrow v$ to be a bump for C.

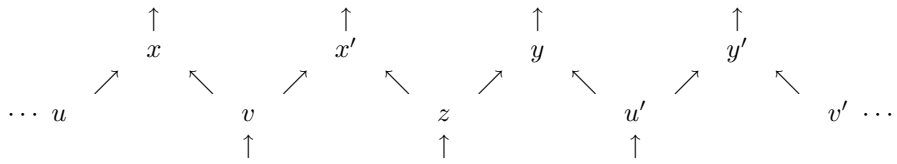

Figure 3: A configuration preventing the detection of a bump-free cycle.

However, we can end up getting a configuration like the one shown in Fig. 3, where, if we first follow the arc $v \to x$ (to avoid the potential bump with apex x), and later (to avoid the potential bump with apex y') we follow $u' \to y'$, then, if in the construction of the cycle we reach vertex z, then we are presented with a dilemma with no solution: if we follow $z \to x'$ (resp. $z \to y$) then we generate a cycle with the bump $\{z, u', y\}$ (resp. $\{z, v, x'\}$) (see Fig. 4).

To overcome this problem, we use a labeling of the arcs of G, based on G's *alternating decomposition*. This labeling will force us to follow a *bump-safe* path during the construction of the cycle C.

An alternating decomposition of G is a partition of the edges of G in alternating paths, of the form $u_1 \leftarrow u_2 \to u_3 \leftarrow u_4 \cdots$ or $u_1 \to u_2 \leftarrow u_3 \to u_4 \cdots$, and alternating cycles, which are closed alternating paths with an even number of edges, like $u_1 \leftarrow u_2 \to u_3 \leftarrow u_4 \to u_1$. We restrict ourself to *maximal* decompositions, in which no path can be further extended.

It is very easy to see that, given the limitation on the in-degree and out-degree of G, such

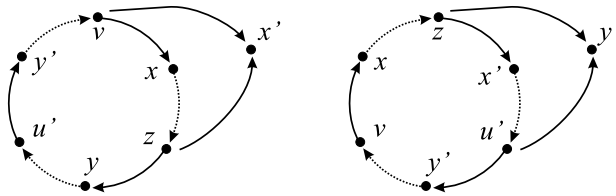

Figure 4: The construction of the cycle introduces either the bump on the left or the one on the right.

a maximal decomposition is unique and can be obtained in a straightforward manner. Figure 5 presents an example of decomposition. Note that the alternating paths do not need to be simple: a vertex can appear twice in the same path. Figure 6 lists the alternating paths for the graph of Figure 5.

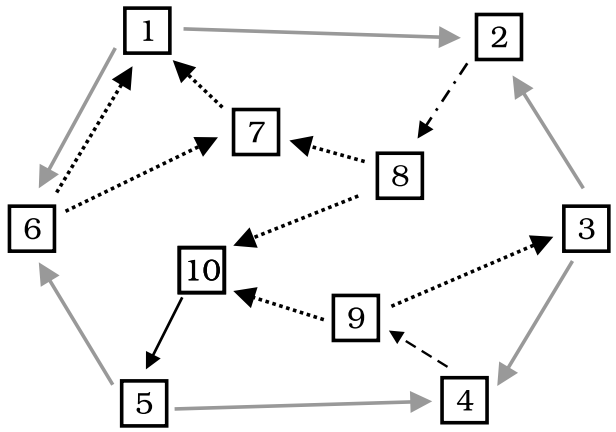

Figure 5: A digraph G with in- and out-degree at most 2, and its five (arc disjoint) alternating paths whose union includes all the arcs of G.

Once we have obtained an alternating decomposition, it is quite intuitive how to avoid the obstacle illustrated in Figs. 3 and 4. For each alternating path or cycle, we consider only the vertices with two outgoing edges. For instance, in the first alternating path of Figure 6, we consider vertices $\{1, 3, 5\}$. We then choose a node, say 3, and mark arbitrarily one of its outgoing edges, say $3 \to 4$. We proceed by marking every other edge in the alternating path, so that each node can be reached by one unmarked edge. An example of such labeling is illustrated in Fig.7. The effect of the labeling is that, if there is a vertex which is the apex of a potential bump, say vertex 4, then we can reach it only through an unmarked edge.

We can now return to our cycle construction. We start from an arbitrary vertex u, and we construct a cycle by following unmarked edges until we reach a node we have already visited.

For example, in the case of Figs. 5 and 7, starting from vertex 1, we obtain the cycle $C = \{1, 6, 7\}$, while starting from vertex 8, we obtain $C = \{8, 10, 5, 4, 9, 3, 2\}$.

This procedure always obtains a cycle, since the labeling guarantees that at least an outgoing arc for every vertex is unmarked. It is also easy to see that the cycle is bump-free. Indeed, suppose that the cycle C obtained by the procedure above has a bump $u \to w \leftarrow v$, with $u, v \in C$ and $w \in G \backslash C$. Since w does not belong to C, while u and v do, then both u and v

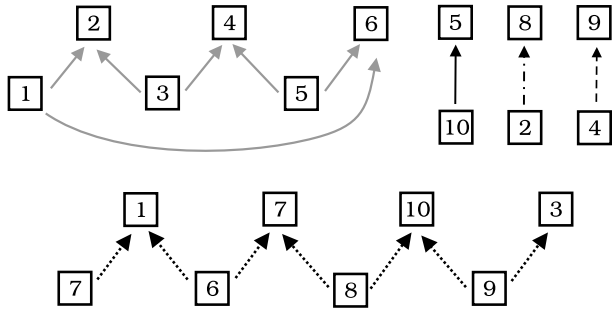

Figure 6: The alternating decomposition of the graph G of Fig. 5.

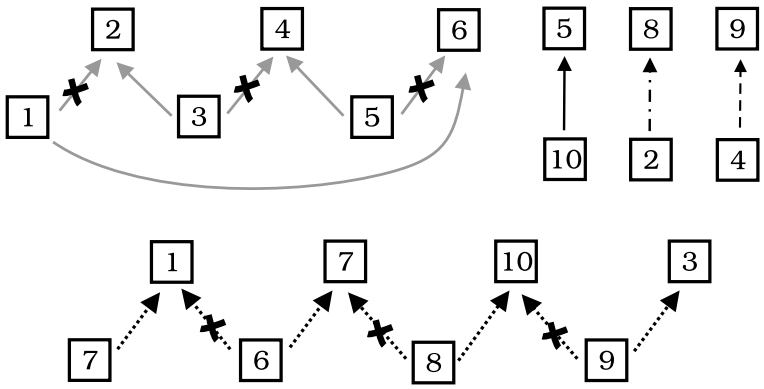

Figure 7: The labeling process, for the graph in Figure 5, based on the alternating decomposition.

must have two outgoing edges, say $x \leftarrow u \rightarrow w \leftarrow v \rightarrow y$, and during the construction of C we must have followed edges $u \rightarrow x$ and $v \rightarrow y$. But this is incompatible with our preliminary alternating labeling since $x \leftarrow u \rightarrow w \leftarrow v \rightarrow y$ is a fragment of an alternating path or cycle.

The subgraph G' to be fed to the second phase of the algorithm will be the subgraph of G induced by the vertices of the bump-free cycle C. It is easy to see that the fact that G' is bump-free implies that we can assign weight 0 to the nodes in $G \backslash G'$ and deal with the weight assignment for G', independently of the rest of the graph.

In the second phase we will find a subgraph G'' of G' such that

- G'' is induced by a cycle;
- G'' is bump-free with respect to G';
- the maximal alternating decomposition of G'' only consists of alternating cycles and alternating paths of length 1, i.e., single arcs.

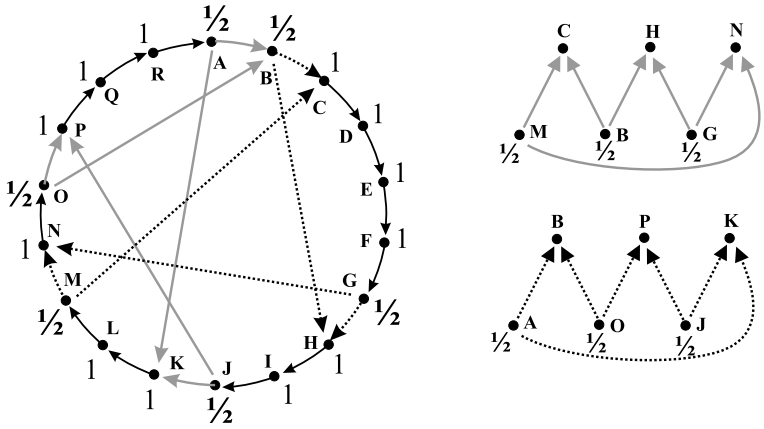

Figure 8: An example of subgraph obtained after phase 2 and the corresponding weight assignment.

These properties imply a "coupling" in G'' between out-degree 2 vertices and in-degree 2 vertices, as shown in Figure 8. What happens is that every in-degree 2 vertex receives its income only from out-degree 2 vertices. It is easy to see that a good assignment can thus be obtained giving weight 1/2 to vertices with out-degree 2, weight 1 to the other vertices in G'' and weight 0 to all the vertices in $G \backslash G''$.

In order to identify a subgraph G'' with these properties, we use the following strategy.

The algorithm, which acts on the current bump-free cycle C and its chords, is divided into a number of stages. At each stage, the algorithm either stops (if the current cycle C induces a subgraph G'' with the desired properties), or returns a shorter cycle C'.

At every stage the algorithm computes an alternating decomposition of the graph induced by the current cycle C. If C does not have chords, or if all the chords belong to alternating cycles as it is the case in Fig. 8, then C and its chords form a subgraph G'' which satisfies our requirements, and the algorithm ends.

Otherwise, we consider any chord (i, j) which is part of an open alternating path (and thus which does not belong to one of the alternating cycles), as in Fig. 9 (top), and process such

chord with the goal of shrinking the current cycle without introducing new bumps. If the edge (h, k), where k immediately follows i along the current cycle, is not present, then we can get a shorter cycle C' by following the path from j to i along C and then closing the path along edge (i, j). The resulting new cycle C' will be bump-free in G since the only potential bump for C' has apex k.

On the other hand, if the edge (h, k) is present, as in the example illustrated in Fig. 9 (bottom), we proceed as follows. If the edge (u, v) (see Fig. 9 – bottom) is not present, then we can again return the cycle C' as shown in the Figure, since v was the only potential bump for C'.

If (u, v) exists, then we repeat the procedure above. Since we are moving along an alternating open path, then we will necessarily reach a chord which can be used to shortcut the cycle without creating a bump.

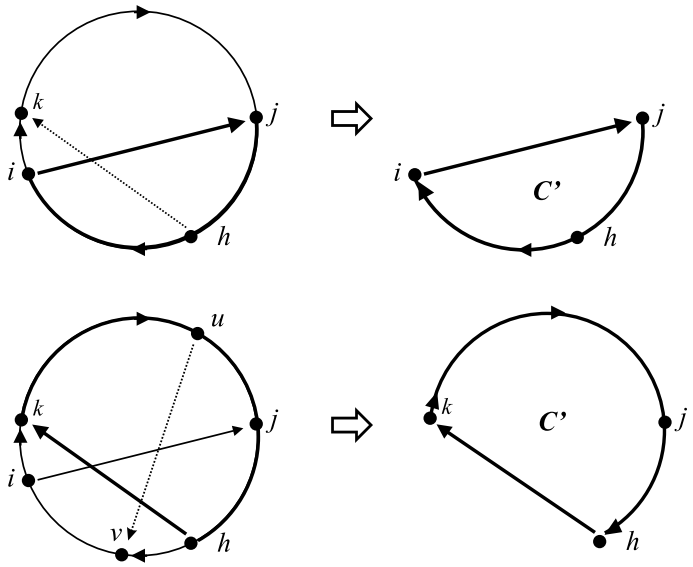

Figure 9: Phase 2: an open alternating path induces a reduction of the current cycle.

Since at every stage we either stop or shrink the current bump-free cycle, the algorithm will always terminate in a finite number of steps, returning a subgraphs with the desired properties. Recalling the example of Fig. 8 and the related discussion, we can now assign positive weights to the vertices of G'', and obtain a good assignment.

4 Pseudocode and correctness

In this section we provide formal definitions, give the pseudocode of the algorithm sketched in Section 3, show its correctness, and prove that it runs in linear time.

Definition 5 [Alternating path] Let $G = (V, A)$ be a directed graph. We say that a sequence of vertices $\langle u_1, u_2, \ldots, u_k \rangle$ defines an *alternating path* if and only if

$$A \supseteq \{(u_1, u_2), (u_3, u_2), (u_3, u_4), (u_5, u_4), \ldots\}$$

or

$$A \supseteq \{(u_2, u_1), (u_2, u_3), (u_4, u_3), (u_4, u_5), \ldots\}.$$

In the following, we will refer to an alternating path by simply listing its (ordered) sequence of arcs.

An alternating path $\langle u_1, u_2, \ldots, u_k \rangle$ may start and end with the same node, but we call it an *alternating cycle* only if, in addition to $u_1 = u_k$, the first and last arc have the same orientation, i.e., if the number of arcs is even. For example, $(2,1), (2,3), (4,3), (4,2), (1,2)$ is a path, while $(1,2), (3,2), (3,4), (1,4)$ is a (simple) alternating cycle.

We say that an alternating path is *maximal* if it cannot be extended.

The paths through a vertex v can be easily computed starting from v and extending the current path in both directions until no further extension is possible.

Proposition 6 *Let $G = (V, A)$ be a strongly connected digraph with in- and out-degree at most 2. Then A can be be uniquely described as the disjoint union A_G of the maximal alternating paths through the vertices of G.*

The partition A_G of Proposition 6 is called the *alternating decomposition* of G and can be easily computed in linear time. Starting from the empty set, we repeatedly pick a node $v \in G$ not yet in A_G, and add to A_G the (at most) two maximal paths through v until no further node remains. Figure 6 shows an example of an alternating decomposition.

Proposition 7 *Every strongly connected digraph $G = (V, A)$ with in- and out-degree at most 2 contains a bump free cycle C.*

The proof of Proposition 7 follows from the informal description of the first phase of the algorithm of the previous section.

Phase 1 In the following we present the pseudocode illustrating the first phase of our algorithm. The procedure stores in a dictionary the vertices visited during the construction of the cycle. The keys to access the dictionary are the vertices being visited; together with a key, the dictionary stores the vertex from which the key has been reached. In this way, once we reach a vertex we have already visited, we can reconstruct the cycle by following the backward edges stored in the dictionary. Note that the starting point of the visit (parameter x) can be any vertex.

```
BUMP-FREE(G, x)
 1  Compute the alternating decomposition A_G of G
 2  for each path P ∈ A_G with P = (u_1, v_1)(u_2, v_2)(u_3, v_3) ... (u_k, v_k)
 3        do if u_i = u_{i+1} then Mark the arc (u_i, v_i), i = 1, ..., k − 1.
 4  Initialize an empty dictionary D
 5  Add the pair (x, nil) to D            ▷ x is the starting point of the visit
 6  u ← x
 7  while true
 8        do Let v be the only vertex such that (u, v) is not marked
 9           if v ∈ D then break
10           Add the pair (v, u) to D
11           u ← v
12  Initialize list C ← ⟨v⟩                          ▷ C holds the cycle's vertices
13  repeat
14           Retrieve the pair (u, w) from D
15           Put u in front of C
16           u ← w
17     until u = v
18  return C
```

Running time of phase 1 The algorithm BUMP-FREE can be implemented in linear time (in the number n of vertices) if the dictionary data structure provides $O(1)$ time insertion and look-up operations. Since the cycle finding process in phase 1 may require to store $\Omega(n)$ vertices, the workspace required by phase 1 is $\Omega(n)$ in the worst-case. This means that if we implement the dictionary using a direct access table (assuming the vertex set is $V = \{0, 1, \ldots, n-1\}$) we can guarantee $O(1)$ access time with no asymptotic penalty in space.

Phase 2 We say that a digraph G with in- and out-degree at most two satisfies the *Decomposition property* if and only if G admits an alternating path decomposition made only of alternating cycles and isolated arcs.

The strategy adopted in Section 3 to find a subgraph G'' satisfying the decomposition property can be quickly turned into a detailed algorithm. However, a naive implementation could easily lead to quadratic running times. The key to obtain a linear time algorithm is the order in which the chords are processed.

Consider the alternating path decomposition $\mathcal{A}_{G'}$ of G' and let $P \in \mathcal{A}_{G'}$ be an open path. Since G' is a cycle C plus a number of chords, it is easy to see that P always starts and ends with an arc of C and that each arc (but the last one) is followed by a chord. It follows that, in P, the arcs have the same orientation while the chords have the opposite orientation. Also, this means that it is always possible to start processing the chords of any open path from its in-degree 1 termination and proceeding towards the out-degree 1 termination. For example, with reference to Figure 9 (left), we first process chord (u, v), then (h, k), and finally (i, j). By doing so, we always produce shorter cycles that are bump-free; moreover, once a path has been completely processed, it need not be considered again. Now, it may be the case that an alternating cycle $C_a \in \mathcal{A}_{G'}$ is not entirely contained in the subgraphs generated by the algorithm. Thus, when processing a chord (x, y), we also check whether any C_a needs to be turned into an open path

because (x, y) "crosses" a chord (u, v) in C_a. If this happens, the open path that results by deleting (u, v) from C_a will be later considered for processing. When no more open paths are left, the induced subgraph of the last generated cycle enjoys the decomposition property.

We present below a detailed pseudocode that implements the above ideas. We assume, without loss of generality, that the vertices of the input graph are numbered from 0 to $n-1$, and that the known cycle C (which is also presented in input to the algorithm) is $0 \to 1 \to 2 \ldots \to n-1 \to 0$. In the pseudocode, we will write $[i]_n$ as a shortcut for $i \bmod n$.

```
GOOD-CYCLE(C, G = (V, A))
 1  n ← |V|
 2  Compute the alternating decomposition A_G of G.
 3  P ← paths in A_G                ▷ Only paths of length > 1 are considered
 4  if P = ∅ then return ⟨C, G⟩
 5  C ← cycles in A_G
 6  while P ≠ ∅
 7        do Pick any ([j_1 − 1]_n, j_1)([j_1 − 1]_n, j_2) ... ([j_{k−1} − 1]_n, j_k)([j_k − 1]_n, j_k) ∈ P
 8           for ℓ ← 1 to k − 1
 9                 do if ([j_ℓ − 1]_n, j_{ℓ+1}) is a chord of C
10                       then Compute the new cycle C'
11                            for u ∈ C\C'
12                                  do if (u, v) ∈ A and v ∈ C'
                                              and {(u, v)} ∈ C_a ∈ C
13                                        then Remove C_a from C
14                                             Place C_a\{(u, v)} in P
15                            C ← C'
16  D ← ∅
17  for C_a ∈ C
18        do if the vertices in C_a belong to C
19              then D ← D ∪ C_a
20  return ⟨C, D⟩
```

Theorem 8 *The cycle C returned by the algorithm* GOOD-CYCLE *has no bump in the input graph G; moreover G_C induced in G by the vertices of C enjoys the decomposition property.*

Proof : We let C_i denote the cycle obtained after i chords of C have been processed by the algorithm GOOD-CYCLE $i = 0, 1, \ldots$ (hence C_0 denotes the cycle in input to the algorithm).
We make a couple of preliminary observations. First of all, by code inspection it follows immediately that no chord is possibly processed twice, since each chord is part of a unique path or cycle in the alternating decomposition. It is also easy to see that, whatever the outcome of the test in line 9, i.e., whether $([j_\ell - 1]_n, j_{\ell+1})$ is a chord of the current cycle C_i or not, $([j_\ell - 1]_n, j_{\ell+1})$ cannot be a chord of any future cycle C_j, $j > i$. We call this the *rule out* property of the processed chords.
A formal proof that C cannot have bumps in G proceeds by induction on the number of chords processed. The base case is obvious, since C_0 is a Hamiltonian cycle of G. Suppose now that C_i does not have bumps and consider C_{i+1}. Clearly, the only interesting case is when $([j_\ell - 1]_n, j_{\ell+1})$ is a chord of C_i (line 9 of GOOD-CYCLE), for otherwise $C_{i+1} = C_i$. Since,

by inductive hypothesis, C_i is bump-free, C_{i+1} has possibly only a bump with apex j_ℓ (see Figure 10), but for this to happen P must include the chord (z, j_ℓ), for some vertex $z \in C_{i+1}$. Since the alternating decomposition contains only maximal paths, it is necessarily the case that $z = [j_{\ell-1}]_n$. However, this implies that such a chord has already been processed, and hence, by the rule out property, cannot be a chord of C_i. Thus C_{i+1} is bump-free.
For a cycle C, we denote by V_C its set of vertices. To prove that the decomposition property holds of G_C, we show that the final cycle C cannot have a chord which is not part of a (complete) alternating cycle of G_C. The proof is by contradiction. Suppose that C has a chord (u, v) which is not part of an alternating cycle of G_C. Now, (u, v) was part of either an open path or a cycle in $\mathcal{A}_G$. If it were part of an alternating path, then we know that (u, v) has been processed at some point, say when the current cycle was C_h, and thus, by the rule out property, cannot be a chord of any cycle C_j, $j > h$. We are thus lead to consider the possibility that (u, v) were part of an alternating cycle C_a and never processed by the algorithm. Since C_a is not an alternating cycle of G_C, there must be a chord (w, z) which is part of C_a such that $w \in V_C$ and $z \in V \backslash V_C$ (or vice-versa). This implies that there must be an index $h \geq 0$ such that $z \in V_{C_h}$ and $z \notin V_{C_{h+1}}$. However, this in turn implies that, after generating C_{h+1}, the algorithm added the (open) path $P_{w,z} = C_a \backslash \{(w, z)\}$ to the set $\mathcal{P}$ of paths still to be processed; since $P_{w,z}$ includes the arc (u, v), this proves that the latter has been processed by the algorithm and hence, again by the rule out property, cannot be a chord of the final cycle. □

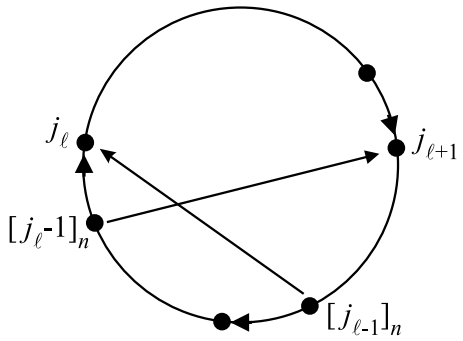

Figure 10: The only "potential" bump for the cycle C_{i+1} has apex j_ℓ.

The correctness of the algorithm Good-Cycle proves the following graph theoretic result.

Corollary 9 *Let G be an Hamiltonian digraph with in- and out-degree at most two. Then there is a cycle C in G such that the subgraph G_C induced by C in G enjoys the decomposition property.*

Running time of phase 2 To prove that the algorithm Good-Cycle runs in time $O(n)$, we implement G using three parallel arrays of size n, namely: F, B, and S. For $i \in \{0, 1, \ldots, n-1\}$, the i-th index positions, $F[i], B[i]$, and $S[i]$ store the following information, respectively:

- the endpoint of the chord out of vertex i (if i has out-degree 2);
- the vertex j such that $F[j] = i$ (if i has in-degree 2);
- the *state* of vertex i.

The state of a vertex i is a value in the set $\{0, 1, 2\}$, where

$$S[i] = \begin{cases} 0 & \text{if } (i, [i+1]_n) \text{ is in the current cycle} \\ 1 & \text{if } (i, F[i]) \text{ is in the current cycle} \\ 2 & \text{otherwise.} \end{cases}$$

Initially, all the vertices are in state 0 (given our assumption on G). We also use a fourth array H, of size n, such that $H[i]$ is a 0/1 flag telling whether the (possibly present) chord $(i, F[i])$ is part of an alternating cycle, $i = 0, 1, \ldots, n-1$. Initially, this information can be easily gathered during the computation of the alternating path decomposition of G, whose running time we already know to be linear in $|V|$.

We now focus on lines 6-19 of the algorithm, where, in the worst case, we consider all the chords of the initial cycle. For any such chord (u, v), we first need to determine whether (u, v) is also a chord of the current cycle. This can be done by simply checking whether the state of u is 0 and that of v is not 2, at the price of two lookups. Altogether, the time spent in lines 6-9 is a constant times the number of chords of I, and thus $O(n)$.

Now, if (u, v) is a chord, we first compute the new cycle (line 10) and then check whether (u, v) intersects any alternating cycle C_a, since then C_a must be turned into an open path (lines 11 through 14). This can be done by:

- changing the state of vertex u from 0 to 1, and,
- for any vertex z which has to leave the current cycle:
  - changing the state of z to 2;
  - checking whether the flags $H[z]$ and $H[B[z]]$ are set, i.e., testing if z is the endpoint of one or two chords that belong to alternating cycles; in the affirmative case, these cycles must be turned into open paths by deleting either $(z, F[z])$ or $(B[z], z)$ (or both) and setting a certain number of H flags to 0 (see Figure 11).

The time spent in lines 11-14 to process a single chord can be $\Omega(n)$, because the number of arcs that leave the current cycle can be linear in n, as is the number of H flags that have to be reset. However, the total time is still $O(n)$ since once an arc leaves the current cycle it never re-enters, and the same is true of an alternating cycle that has been turned into a path.

When $\mathcal{P}$ is emptied, the alternating cycles in $\mathcal{C}$ (if any remains) are either completely inside C or completely outside C. As a consequence, line 18 requires constant time (it is sufficient to check a single chord of any cycle C_a) and the time spent in lines 16-20 is proportional to the number of cycles, thus again $O(n)$.

The good assignment Let C be the cycle returned by phase 2 of the algorithm and let G_C be the corresponding induced subgraph of G. If the alternating path decomposition of G_C is composed only of single arcs, i.e., if the set D returned by GOOD-CYCLE is empty, then $G_C = C$. In this case a good assignment gives weight 1 to the vertices of C and 0 to those in $G \backslash C$. This guarantees that all the vertices in C have income 1, and, since C is bump-free in G, no vertex in $G \backslash C$ can have a higher income.

Now suppose that $G_C \neq C$. This means that the alternating decomposition of G_C contains at least an alternating cycle. The property that guarantees that G admits a good assignment is summarized in Lemma 10.

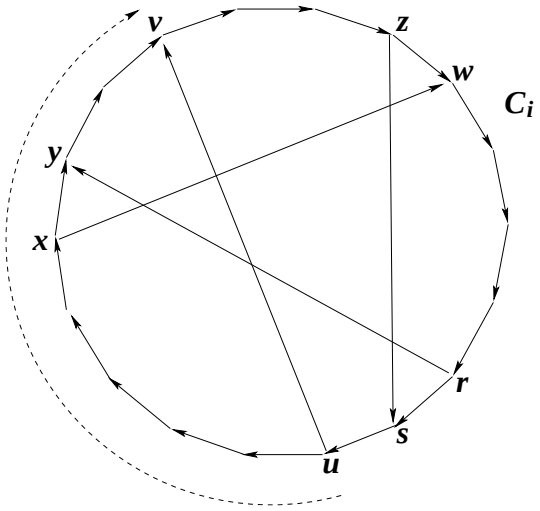

Figure 11: (u, v) is a chord for the current cycle C_i. The state of all the vertices that will not be part of C_{i+1} (in the circle arc indicated by the dotted curve) is assigned value 2. Also, since the flag $H[x]$ is set, the alternating cycle $(x, y)(r, y)(r, s)(z, s), (z, w)(x, w)$ is turned into a path by removing the arc (x, w) and by resetting the flags $H[x], H[z]$, and $H[r]$. Note, then, that when vertex y is considered, the flag $H[B[y]] = H[r]$ has been already reset and thus nothing is done besides assigning the new value to $S[y]$.

Lemma 10 *Let $G = (V, A)$ be a Hamiltonian digraph with in- and out-degree at most two, and suppose that G satisfies the decomposition property. Then, all the vertices with in-degree 2 get their input from vertices with out-degree 2.*

Proof : Since G is Hamiltonian, its vertices can be arranged in a circle and renamed from 0 to $|V| - 1$. Now, suppose that a vertex v has in-degree 2. It follows that the input to v comes from $u = [v - 1]_{|V|}$ and from some other vertex $w \neq u$. Clearly (w, v) must be a chord, and this implies that w has out-degree 2. However, it also means that in G there exists the sequence of arcs $(u, v)(w, v)(w, [w + 1]_{|V|})$. By Corollary 6, such sequence necessarily belongs to the same alternating path. Since G satisfies the decomposition property, this path is actually a cycle. It follows that there must be another arc (u, z) in the cycle, for some $z \neq v$. Thus u has out-degree 2 as well. □

Using the result of Lemma 10, we can easily find a good assignment for G. We give weight 1/2 to all the vertices in G_C with out-degree 2 and weight 1 to the vertices with out-degree 1. Lemma 10 guarantees that all the vertices in G_C receive income 1. As before, the assignment is completed by giving weight 0 to the vertices in $G \backslash C$. The complete algorithm is reported below.

GOOD-ASSIGNMENT(G)

```
1  C ← BUMP-FREE(G, 1)                          ▷ Assume V = {1, 2, ..., n}
2  Rename the vertices of C from 0 to |C| − 1
3  G_C ← graph induced by C in G
4  ⟨C, D⟩ ← GOOD-CYCLE(C, G_C)
5  H ← graph induced by C in G_C
6  for any alternating cycle C_a in D
7        do for any vertex v ∈ C_a              ▷ v has out-degree 2 in H
8              do Give weight 1/2 to v
9  Give weight 1 to the remaining vertices in C
10 Give weight 0 to all the vertices in G\C
```

The results of this section can be summarized as follows.

Theorem 11 *Let G be a digraph on n vertices, with in- and out-degree at most two. A good assignment to G can be computed in $O(n)$ time. Moreover, the weights can be chosen from the set $\{0, 1/2, 1\}$.*

Building upon the Lemmas of Section 2, Theorem 11 immediately leads to Corollary 12 below.

Corollary 12 *Let A and B be $m \times n$ zero-one matrices with at most two nonzero entries per row and column. A Nash equilibrium for the bimatrix game (A, B) can be computed in $O(n+m)$ time. Moreover, the entries of the Nash equilibrium strategies can be chosen from the set $\{0, 1/2, 1\}$.*

5 Conclusions

The problem of computing a Nash equilibrium for 2-player win-lose games is complete for the class $PPAD$, and thus unlikely to be solvable in polynomial time. The core of the computational difficulty is in finding the support of a Nash equilibrium. In fact, once the support is known, the problem simplifies to linear programming.

In this paper we have dealt with a restriction under which the determination of the support translates into an interesting problem on certain very sparse directed graphs. By taking advantage of the structural properties that arise from the sparseness, we have been able to devise an efficient algorithm which determines the support, and then the actual weights.

Future work includes further exploring and possibly extending the frontier of tractability of the problem, e.g., by mitigating the sparsity assumptions.

References

[1] T. Abbott, D. Kane, P. Valiant, On the Complexity of Two-Player Win-Lose Games. Proc. 46th Annual IEEE Symposium on Foundations of Computer Science, pp. 113-122 (2005).

[2] X. Chen, X. Deng, 3-NASH is PPAD-Complete, ECCC TR05-134 (2005).

[3] X. Chen, X. Deng, Settling the Complexity of 2-Player Nash-Equilibrium, ECCC TR05-140 (2005).

[4] B. Codenotti, D. Stefankovic, On the computational complexity of Nash equilibria for (0,1)-bimatrix games. Information Processing Letters, 94(3), pp.145-150 (2005).

[5] C. Daskalakis, P. Goldberg, C. Papadimitriou, The complexity of computing a Nash equilibrium, ECCC TR05-115 (2005).

[6] C. Daskalakis, C. Papadimitriou, Three-Player Games Are Hard, ECCC TR05-139 (2005).

[7] P. W. Goldberg, C. Papadimitriou, Reducibility Among Equilibrium Problems, ECCC TR05-090 (2005).

[8] C.E. Lemke and J.T. Howson, Equilibrium points in bimatrix games, Journal of the Society for Industrial and Applied Mathematics 12, pp. 413- 423 (1964).

[9] J. Nash, Non-Cooperative Games, Annals of Mathematics 54(2), pp. 286-295 (1951).

[10] C. Papadimitriou, On the Complexity of the Parity Argument and other Inefficient Proofs of Existence, Journal of Computer and System Sciences 48, pp. 498-532 (1994).

[11] R. Savani and B. von Stengel, Exponentially Many Steps for Finding a Nash Equilibrium in a Bimatrix Game. Proc. 45th Annual IEEE Symposium on Foundations of Computer Science, pp. 258-267 (2004).

ECCC ISSN 1433-8092
http://www.eccc.uni-trier.de/eccc
ftp://ftp.eccc.uni-trier.de/pub/eccc
ftpmail@ftp.eccc.uni-trier.de, subject 'help eccc'