From game trees to game graphs

W.T.M. Kars

RUU-CS-90-43
December 1990

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

From game trees to game graphs

W.T.M. Kars

Technical Report RUU-CS-90-43
December 1990

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN:0924-3275

From game trees to game graphs

W.T.M. Kars
Department of Computer Science, Utrecht University,
P.O. Box 80.089, 3508 TB Utrecht, the Netherlands.

Abstract

We extend the definition of the score of a game tree using the minimax rule to game graphs containing cycles. This is accomplished by taking the limit of a sequence of scores of game trees that in some sense approximate the game graph. We show that this score is well-defined for all game graphs, and give algorithms for its evaluation, including one that uses memoization.

1 Introduction

In this note we consider games for two players, called Max and Min, who take alternating turns to move. Both players have perfect knowledge of the game and its current position and the game is deterministic in that there is no chance involved. Examples of the sort of game we have in mind are chess and the L-game of [de Bono] (cf. also [Berlekamp, Conway & Guy]). Games are usually represented by a *game graph*, i.e. a directed graph whose vertices represent the possible positions of the game and whose edges represent the possible moves. Algorithms for finding an optimal move involve the generation of (part of) the game graph, evaluating the leaves of this graph using some scoring function and backing up the scores to the root of the graph. The scoring function assigns values to the leaves taken from some ordered domain. The higher a score, the better it is for Max and the worse it is for Min. Backing up the score to the root involves a minimax procedure: take the maximum of all subsequent scores if it's Max's move, otherwise take the minimum. This reflects the fact that a player will do the best possible move, assuming the other player will do the same and that both have perfect knowledge. Usually this calculation is performed in a depth-first search (dfs) way.

All this is well-known and well-defined if the game graph is a tree, but what happens when the graph contains semicycles or cycles?[1] Proper semicycles occur if a position may be reached by more than one sequence of moves (without returning to a previous position). They pose no problem: the score of a node is still defined uniquely by the minimax rule. In fact, the sharing of nodes offers the opportunity to save work if memoization (a term taken from functional programming) is used, i.e. if the scores of (shared) nodes are retained. This eliminates the need for recalculation of a score when a different path to a shared node is followed by the dfs-algorithm.

Cycles occur if there is a sequence of moves leading back to a previous position. This implies the possibility of non-termination (divergence). The rules of the game will have to deal with this possibility. We will assume the simple and fair rule: divergence = draw.

[1] The terminology is taken from [Harary]: both semicycles and cycles are closed sequences of edges. The edges in a cycle point in the same direction, whereas in a semicycle this is not necessary. Stated differently: a cycle $\in E^+$ and a semicycle $\in (E \cup E^{-1})^+$. A proper semicycle will be a semicycle that is not a cycle.

A crude way to handle cycles would be to consider the node that closes a cycle to be a terminal node and evaluate it as such. However, we can do better than that, as we will show.
Section 2 contains the basic definitions of the types of graphs that will be used. Section 3 illustrates the main problem by an example. Section 4 contains the definition of the minimax score for any game graph by a limit procedure. In section 5 we prove that this limit converges in finite time for all finite game graphs, leading to a distributed algorithm for its calculation. In section 6 it is shown that the score of a finite game graph may be computed by a simple dfs-algorithm. Section 7 considers the problems raised if one wants to apply memoization, giving rise to a more efficient algorithm for the evaluation of the score of a finite game graph, and section 8 concludes with some final remarks.

The results appear to be new. Apart from [Boffey], who only handles proper semicycles, the only work on extending the evaluation of game trees to game graphs seems to be [Gijlswijk et al.] (see section 8). The monumental work by [Berlekamp, Conway & Guy] may contain a treatment in some way, but the idiosyncratic notation and terminology makes it hard to compare it to the present work.

2 Definitions and notation

The two most important types of structure that we need are the finite game graphs and the infinite game trees.

Definition 2.1

a $\mathbf{G}_\omega$ is the class of finite-branching, connected, rooted, directed graphs with no more than ω (i.e. countably infinite) nodes. $G \in \mathbf{G}_\omega$ if $G = (V, E, r)$ with
- V is a set of nodes (vertices), $|V| \leq \omega$,
- $E \subseteq V \times V$ is the set of directed edges,
- $r \in V$ is the root,
- for every $v \in V$ there is a (directed) path from r to v,
- for every $v \in V : |E\ v| < \omega$, where $E\ x = \{y \in V \mid (x, y) \in E\}$.

b $\mathbf{G}$ is the subclass of $\mathbf{G}_\omega$ with a finite number of nodes. $\mathbf{T}_\omega$ is the subclass of $\mathbf{G}_\omega$ of (infinite) trees and $\mathbf{T}$ the subclass of $\mathbf{G}$ of finite trees.

$$\begin{array}{ccc} \mathbf{G}_\omega & \supset & \mathbf{G} \\ \cup & & \cup \\ \mathbf{T}_\omega & \supset & \mathbf{T} \end{array}$$

c For any class of graphs $\mathbf{X}$, the class of *game graphs of* $\mathbf{X}$, denoted by $\mathbf{GX}$, is the subclass of $\mathbf{X}$ of graphs which are
- bipartite, i.e. there is a partition of V: $V = V_- \cup V_+$, where $V_- \cap V_+ = \emptyset$ and such that $E \subseteq (V_- \times V_+) \cup (V_+ \times V_-)$, and

- provided with a (scoring) function on the terminal nodes (leaves), $t : F \to K$, where $F = \{x \in V \mid E\ x = \emptyset\}$ is the set of leaves and $(K, \leq)$ is a finite ordered set containing 0 (representing a draw), assumed fixed throughout this note.

Unless stated differently, a game graph will be normalized, in the sense that $r \in V_+$ (Max begins).

For any game graph $G = (V, E, r, t)$, $V_\uparrow = V_+ \setminus F$ is the set of maximizing nodes (Max's turn to play) and $V_\downarrow = V_- \setminus F$ is the set of minimizing nodes (Min's turn to play).

Two game graphs are isomorphic ($\cong$) if there is a bijection between the nodes of the two graphs which preserves adjacency, roots and t-scores.

□

We will use some notation of the Bird-Meertens formalism (see e.g. [Bird]) and of functional programming in general. In particular, $\cdot$ denotes apposition (i.e. function *ap*plication or function com*position*, depending on the context). $*$ denotes the *map* operator, which takes a function and a list and applies the function to every element of the list. $/$ denotes the *reduce* operator, which takes an associative binary operator and a list and returns the value of the expression formed by putting the operator between the elements of the list. $\uparrow$ denotes the binary maximum operator and $\downarrow$ denotes the binary minimum operator.
Properties of $\uparrow$ and $\downarrow$ that will be needed: both are commutative, associative and idempotent ($a \uparrow a = a$) and they distribute over each other.

An overview of the notation may be found at the end of this note.

3 Problem statement

Given a game graph $G = (V, E, r, t) \in \mathbf{GG}$, the problem consists of (uniquely) extending t to a function s (the score) on V:

$$x \in F \to s\ x = t\ x \qquad \text{(T)}$$

subject to the minimax rule:

$$x \in V_\uparrow \to s\ x = \uparrow / \cdot s* \cdot E\ x \qquad \text{(MM)}$$

$$x \in V_\downarrow \to s\ x = \downarrow / \cdot s* \cdot E\ x$$

In the sequel this will usually be abbreviated by $x \in V_\updownarrow \to s\ x = \updownarrow / \cdot s* \cdot E\ x$. Note that $\uparrow / \cdot s* \cdot E\ x$ should be parsed as $(\uparrow /) \cdot (s*) \cdot (E\ x)$. We will also refer to the score of a game graph, when we mean the score of its root.

Theorem 3.1 If $G \in \mathbf{GG}$ is cycle-free, there is a unique solution s to (T+MM).

Proof Straightforward bottom-up calculation.
□

If G is not cycle-free, the score may not be determined uniquely by (T+MM).

Example 3.2 See the graph shown in figure 1.

The score (s-value) of a node will be denoted by the name of the node itself. The condition of the minimax-rule is

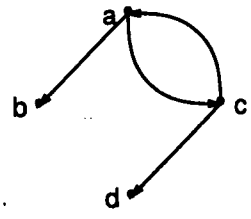

Figure 1: Example of a graph with a cycle; $a \in V_+, c \in V_-$.

$$(a = b \uparrow c) \ \wedge \ (c = d \downarrow a)$$

which is equivalent to

$$(a = b \geq c = d) \ \vee \ (d \geq a = c \geq b)$$

This means that s is uniquely determined if $b \geq d$, since then $a = b$ and $c = d$. This is obvious if one considers the strategy followed by Max in a: Max will certainly not move to c, since this will be followed by a move to d, leaving Max worse off.

However, if $b < d$, the minimax-rule only restricts the scores of a and c to be equal and inside the interval $[b, d]$. Examination of the possible move sequences shows the following. If Max moves from a to c, Min could either 'offer a draw' by moving back to a or choose d. If $d > 0$, Min should play a: Max may choose b, which is better for Min than d, or Max chooses c again which will result in a draw with score 0, which is also better for Min than d. However if $d < 0$, it is better to play d rightaway. Moving to a will certainly not result in Max playing to b, since $b < d < 0$, but Max will play c again. In short, in c, Min will choose d if it is smaller than 0, otherwise it chooses a. Following this line of reasoning gives the following scores, which are valid in all cases:

$$\begin{aligned} a &= b \uparrow (d \downarrow 0) \\ c &= d \downarrow (b \uparrow 0) \end{aligned}$$

□

For this particular case, we see that the score of a —taking into account the possibility of a draw— may be calculated in a straightforward depth-first search (dfs) way, if the score of a 'back link' in the dfs is set to 0. In section 6 we will prove that this is true in general.

4 Definition of s for general game graphs

The score of a game graph will be defined as the score of the (possibly infinite) isomorphic tree that is obtained by 'unraveling' the proper semicycles and 'unwinding' the cycles. It is the tree that contains all paths in the graph in a 1-1 way. See figures 2 and 3 for examples of this transformation.

The definition of the minimax score of (the root of) infinite trees should be subject to the requirement that

- the score of a finite tree equals the usual one, i.e. it is a consistent extension, and
- the tree resulting from unwinding a simple cycle is assigned a score of 0.

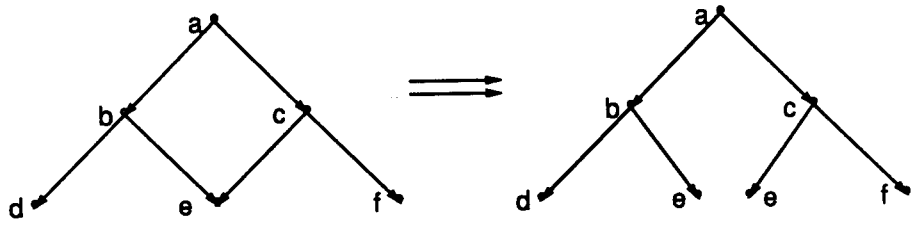

Figure 2: Example of unraveling a graph with a proper semicycle.

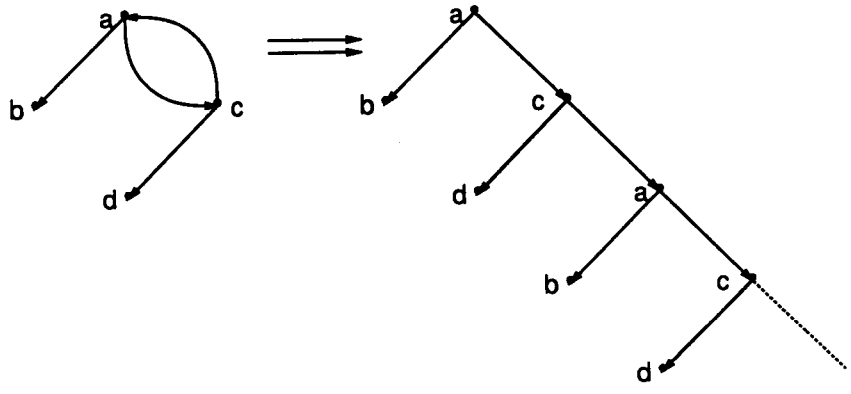

Figure 3: Example of unwinding a graph with a cycle.

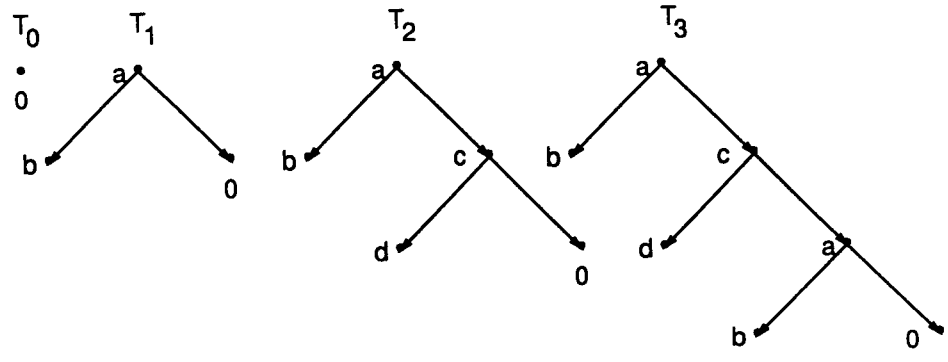

Figure 4: Approximations of the tree of fig. 3.

This is accomplished by defining the score of an infinite tree to be the limit of the scores of finite approximations of the tree. The nth approximation consists of that part of the tree that is reachable from the root in n steps, and where fresh leaves, i.e. those leaves of the approximation that are not leaves of the original tree, are assigned a score of 0. See figure 4 for an example.

Interpreting this definition in terms of the original graph, we get the following

Definition 4.1 For $(V, E, r, t) \in \mathbf{GG}_\omega$ let

$$
\begin{array}{lcl}
s\ r & = & \lim_{n\to\infty} s_n\ r \\
\textbf{where} & & \\
s_0\ p & = & 0 \\
s_{n+1}\ p & = & \textbf{if } p \in F \to t\ p \\
 & & \textbf{elif } p \in V_\updownarrow \to \updownarrow / \cdot s_n * \cdot E\ p \\
 & & \textbf{fi}
\end{array}
$$

□

The limit in this definition exists for all game graphs, as will be shown in the next section.

The definition is evidently in accordance with the usual one when the graph is cycle-free. Furthermore, assigning score 0 to fresh leaves ensures that a simple draw will be assigned a score of 0 (see figure 5). Hence our requirements are fulfilled.

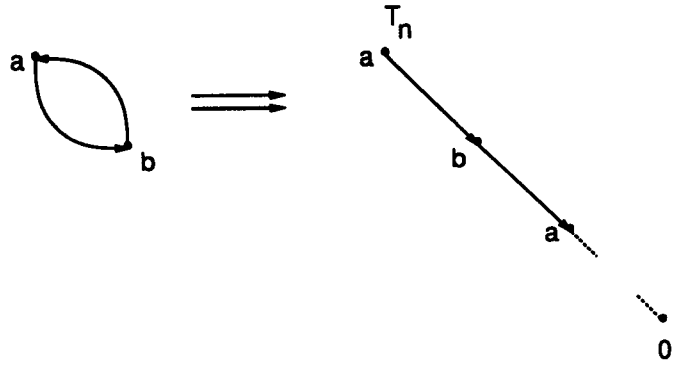

Figure 5: Approximations of a simple cycle.

5 Convergence of s

In this section we will show that the limit $\lim_{n\to\infty} s_n$ exists and, for finite game graphs, converges in $O(|V|)$ steps, each taking $O(|E|)$ time.

Let $G = (V, E, r, t) \in \mathbf{GG}$. By the definition of s_n

$$
\begin{array}{lcl}
s_0 & = & 0^V \\
s_{n+1} & = & A\ s_n
\end{array}
$$

for some operator $A : K^V \to K^V$ independent of n.

Definition 5.1 Define a partial order on K:

$$a \leq_1 b \equiv (b \leq a < 0) \vee (a = 0) \vee (0 < a \leq b)$$

Define a partial order on K^V:

$$x \leq_V y \equiv \forall v \in V : x \cdot v \leq_1 y \cdot v$$

□

It is straightforward to prove that $\leq_1$ and $\leq_V$ are partial orders, i.e. that they are reflexive, antisymmetric and transitive.

Lemma 5.2 $(K^V, \leq_V)$ is a complete partial order (cpo) with least element 0^V.

Proof Recall that a cpo is a poset with a least element and in which every ascending chain has a least upperbound. An ascending chain in a poset $(P, \sqsubseteq)$ is a countable sequence of elements of P: $\langle a_n \rangle_{n=0}^{\infty}$, such that for all $i \geq 0$: $a_i \sqsubseteq a_{i+1}$.
Since the poset $(K^V, \leq_V)$ is the Cartesian product of $|V|$ copies of $(K, \leq_1)$, it is sufficient to prove that $(K, \leq_1)$ is a cpo (cf. [Loeckx & Siebert], p. 75).
It is easy to see that K is a cpo; for all $s \in K^V$: $0 \leq_V s$, so 0 is the least element; also, since K is finite, every ascending chain has a least upperbound.
□

Remark: let $K_- = \{a \in K \mid a \leq 0\}$ and $K_+ = \{a \in K \mid a \geq 0\}$, then $(K, \leq_1)$ is just the so-called coalesced sum of the cpos $(K_-, \geq)$ and $(K_+, \leq)$.

Next we show that s_n is a monotone sequence in K^V. This amounts to proving that, if each value in a set (viz. $s_n * \cdot E\ v$) 'moves away' from 0, then this also holds for the maximum and minimum of these values (viz. $s_{n+1}\ v$). Note that it is perfectly possible for a maximizing node ($\in V_\uparrow$) to have a *decreasing* (in the sense of $\leq$) sequence of approximate scores.

Remark: hence, if we define a partial order $\leq_{GT}$ on $\mathbf{GT}_\omega$ (incl. the empty tree) by $T \leq_{GT} T'$ iff $T = \emptyset$ or T is an approximation of T' (see section 4), then it is not difficult to see that $(\mathbf{GT}_\omega, \leq_{GT})$ is a cpo with least element $\emptyset$, and (defining $s\ \emptyset = 0$) that s is an order-continuous function: $(\mathbf{GT}_\omega, \leq_{GT}) \to (K, \leq_1)$.

Lemma 5.3

$$\forall n \in \mathbf{N} : s_n \leq_V s_{n+1}$$

Proof By induction on n.
Let $P_n(v) = (s_n\ v \leq_1 s_{n+1}\ v)$, and $P_n = s_n \leq_V s_{n+1}$, so $P_n = \forall v \in V : P_n(v)$.

Base: P_0 holds, since $s_0 = 0^V$, which by lemma 5.2 is the least element of the cpo $(K^V, \leq_V)$.

Step: Assume P_k holds for all $k < n$, we will prove that P_n holds. Let $v \in V$.

- $v \in F$: Then $s_0\ v = 0$ and for all $n > 0 : s_n\ v = t\ v$ is constant. Hence $P_n(v)$ holds.
- $v \in V_\uparrow$: For $k \in \mathbf{N}$ let l_k be a successor of v with a maximal value of s_k, i.e. $l_k \in E\ v$ and $\forall w \in E\ v : s_k\ w \leq s_k\ l_k\ (= s_{k+1}\ v)$.

  Note that $P_k(v) = Q(s_k\ v, s_{k+1}\ v)$ where

$$\begin{aligned} Q(a,b) &= (b \leq a < 0) \vee (a = 0) \vee (0 < a \leq b) \\ &= (a < 0 \to b \leq a) \wedge \\ &\quad (a = 0 \to \mathbf{true}) \wedge \\ &\quad (a > 0 \to a \leq b) \end{aligned}$$

so the proof of $P_n(v)$ will consider these 3 cases.
The following equations will be useful

$$Q(a,b) \wedge (a < 0) = (0 > a \geq b) \quad \text{(A)}$$
$$Q(a,b) \wedge (a > 0) = (0 < a \leq b) \quad \text{(B)}$$

1. $s_n\ v < 0$. Proof obligation: $s_n\ v \geq s_{n+1}\ v$.
   First:

   $$\begin{array}{lll} s_{n-1}\ l_n & \leq & \text{"def. of } s_n\text{"} \\ s_n\ v & < & \text{"assumption"} \\ 0 & & \end{array}$$

   so by (A) using $P_{n-1}(l_n)$, which holds by the induction hypothesis:

   $$s_n\ l_n \leq s_{n-1}\ l_n \quad (*)$$

   Then:

   $$\begin{array}{lll} s_n\ v & \geq & \text{"def. of } s_n\text{"} \\ s_{n-1}\ l_n & \geq & \text{"(*)"} \\ s_n\ l_n & = & \text{"def. of } l_n\text{"} \\ s_{n+1}\ v & & \end{array}$$

2. $s_n\ v = 0$. No proof obligation.
3. $s_n\ v > 0$. Proof obligation: $s_{n+1}\ v \geq s_n\ v$.
   First:

   $$\begin{array}{lll} s_{n-1}\ l_{n-1} & = & \text{"def. of } l_{n-1}\text{"} \\ s_n\ v & > & \text{"assumption"} \\ 0 & & \end{array}$$

   so by (B) using $P_{n-1}(l_{n-1})$, which holds by the induction hypothesis:

   $$s_n\ l_{n-1} \geq s_{n-1}\ l_{n-1} \quad (**)$$

   Then:

   $$\begin{array}{lll} s_{n+1}\ v & \geq & \text{"def. of } s_{n+1}\text{"} \\ s_n\ l_{n-1} & \geq & \text{"(**)"} \\ s_{n-1}\ l_{n-1} & = & \text{"def. of } l_{n-1}\text{"} \\ s_n\ v & & \end{array}$$

- $v \in V_1$: The proof goes analogous to the previous case.

□

Theorem 5.4

$\langle s_n \rangle_{n=0}^{\infty}$ converges in $O(|K| \times |V|)$ steps.

Proof

By lemma 5.3 $\langle s_n \rangle_{n=0}^{\infty}$ is an ascending chain, which by lemma 5.2 has a least upperbound. Furthermore, since $s_{n+1} = A\ s_n$, as soon as $s_{k+1} = s_k$ for some k, then $s_{k+i} = s_k$ for all $i \geq 0$. This means that the slowest rate of convergence is obtained if only one component of s_n

changes value to the next higher value in K. Since the longest ascending chain in K has length $|K_-| \uparrow |K_+| \leq |K|$, convergence takes $\leq |K| \times |V|$ steps.
□

Note that each step takes $O(|E|)$ time!

The upperbound for the number of steps until convergence may be sharpened somewhat by observing that the possible values of any $s_k\ v$ are taken from the set $(t\ F) \cup \{0\}$, since the maximum and minimum of a finite set are contained in that set.

This leads to a straightforward (synchronous) distributed algorithm for the computation of s. Consider a processor network with the same topology as the game graph: for every $v \in V$ there is a processor p_v and for every $(v, w) \in V$ there is a communication link from p_w to p_v (order reversed!). Every processor p_v maintains the current value of $s_n\ v$ in a variable S, initialized to 0. Processor p_v repeatedly executes a loop, in which it sends the current value of S to its parent, awaits the S-values of all its children and recomputes S by taking the maximum (if $v \in V_\uparrow$) or minimum (if $v \in V_\downarrow$) of the children's values. A 'leaf' processor p_v ($v \in F$) repeatedly sends the value $t\ v$ to its parent after the first step. This algorithm should be completed with a stability detection algorithm. We will not pursue this further, but direct our attention to sequential algorithms.

6 Dfs-algorithm

In this section we will show that the score of a finite game graph may be evaluated by a straightforward depth-first search algorithm, in which a recurring position is assigned score 0. A recurring position is a position which is already on the current path from the root.

First some notation: for $T \in \mathbf{GT}_\omega$, $P\ T$ means that P is a subtree of T. A subtree will always be a *maximal* subtree in the sense that it contains all nodes reachable from its root. For $n \in K$, $\hat{n}$ denotes a game tree consisting of a leaf with t-value n, i.e. $\hat{n} = (\{r\}, \emptyset, r, \lambda r.n)$. If $P\ T$ and $Q \in \mathbf{GT}_\omega$, then $T[Q/P]$ denotes the result of replacing P by Q in T.

A basic observation states that in order to compute the score of a tree, a subtree may be replaced by its score: if $P\ T \in \mathbf{GT}_\omega$, then $s\ T = s\ T[\widehat{s\ P}/P]$. This will be derived from a more general lemma, for which some extra definitions and notation is introduced.

Definition 6.1
The binary relation $=_s$ is defined on $\mathbf{GT}_\omega$ as the 'equivalence kernel of s', i.e.

$$P =_s Q \iff s\ P = s\ Q$$

□

Lemma 6.2 $=_s$ is an equivalence relation.

Proof It is easy to prove that $=_s$ is reflexive, symmetric and transitive.
□

Lemma 6.3 $P =_s \widehat{s\ P}$.

Proof Straight from the definitions of $\widehat{\ }$ and $=_s$.
□

Now we introduce contexts, which will allow us to state the substitution result mentioned before in a more symmetrical way. A *context* $C[\,]$ is a tree with a 'hole'; it may be conceived of as a tree in which a subtree is replaced by some marker. An *empty context* is just a hole. $C[P]$ denotes the result of replacing the marker in $C[\,]$ by a tree P. $\mathcal{C}_\omega$ denotes the class of all contexts of trees in $\mathbf{GT}_\omega$.

Lemma 6.4 [Substitution property]
For $P, Q \in \mathbf{GT}_\omega$ and $C[\,] \in \mathcal{C}_\omega$

$$P =_s Q \;\Rightarrow\; C[P] =_s C[Q]$$

Proof By the definition of s there exists, for every $n \in \mathbf{N}$, a function $C_n : K \to K$, such that for all $T \in \mathbf{GT}_\omega : s_n\ C[T] = C_n\ (s_{n \ominus d}\ T)$, where d is the depth of the hole in $C[\,]$ and $n \ominus d = 0 \uparrow (n - d)$ (this may be proved by induction).

Since K is finite, existence of the limit $\lim_{n \to \infty} s_n\ C[T]$ means that $\langle s_n\ C[T] \rangle_{n=0}^{\infty}$ is eventually constant, i.e. $\forall T \in \mathbf{GT}_\omega : \exists N_T \in \mathbf{N} : \forall n \geq N_T : s_n\ C[T] = s\ C[T]$.

Let $N = (N_P \uparrow N_Q) + d$, then for all $n \geq N$

$$s_n\ C[P] = C_n\ (s_{n \ominus d}\ P) = C_n\ (s\ P) = C_n\ (s\ Q) = C_n\ (s_{n \ominus d}\ Q) = s_n\ C[Q]$$

from which the statement of the lemma follows.
□

Together, lemma 6.2 and lemma 6.4 show that $=_s$ is a congruence with respect to 'tree constructors'.

Lemma 6.5 [Special substitution property]
Let $P, Q \in \mathbf{GT}_\omega$ and $C[\,] \in \mathcal{C}_\omega$. Then

$$C[P] =_s C[\widehat{s\ P}]$$

Proof Use lemma 6.3 in the substitution lemma.
□

Using the substitution property we are able to prove that a subtree isomorphic as a game graph to a proper subtree may be replaced by a leaf with t-score 0. This is precisely the situation that occurs if the tree results from unwinding a graph containing a cycle.

Theorem 6.6 [Cutoff theorem]
Let $P \in \mathbf{GT}_\omega$ and $Q[\,], R[\,] \in \mathcal{C}_\omega$ with $Q[\,]$ non-empty. Then

$$Q[P] \cong P \;\Rightarrow\; R[Q[P]] =_s R[Q[\widehat{0}]]$$

Proof (In 'substitution notation' this reads: Let $R \in \mathbf{GT}_\omega$ and $P\ \ Q\ \ R$ with $P \cong Q$. Then $R =_s R[\widehat{0}/P]$).

It suffices to prove that for all $P \in \mathbf{GT}_\omega$ and $Q[\,] \in \mathcal{C}_\omega$ with $Q[\,]$ non-empty,

$$Q[P] \cong P \;\Rightarrow\; Q[P] =_s Q[\widehat{0}]$$

since the result then follows by the substitution property.

Let $T = Q[P]$. Let $\dot{T} = p_0, \ldots, p_d = \dot{P}$ be the path from the root of T to the root of P. By the special substitution property the subtrees of p_j $(j \neq d)$ that are not on this path (i.e. that are

not rooted at p_{j+1}) may be replaced by their score. The resulting tree will still be denoted by T. Let $a_j = \textbf{if } p_j \in V_\updownarrow \rightarrow \updownarrow/\{s\ S \mid S\ \ p_j, p_{j+1} \notin S\}\ \textbf{fi}$, $j = 0, \ldots, d-1$.

Claim: $s_d\ T = s_{d+1}\ T$.
Proof sketch of claim (the full proof may be found in the appendix): Assume WLOG that $p_0 \in V_\uparrow$. Then $s_d\ T = a_0 \uparrow (a_1 \downarrow \cdots (a_{d-1} \downarrow 0))$ and, since $T \cong P$, $s_{d+1}\ T = a_0 \uparrow (a_1 \downarrow \cdots (a_{d-1} \downarrow (a_0 \uparrow 0)))$. Using distributivity and idempotency of $\uparrow$ and $\downarrow$, it may be shown that the second occurrence of "$a_0 \uparrow$" may be 'absorbed' by the first one. (End of Proof sketch)

This result —with $\uparrow$ and $\downarrow$ interchanged— may be applied to the tree rooted at p_1, again using that $T \cong P$, thereby proving that $s_d\ p_1 = s_{d+1}\ p_1$ holds, and therefore $s_{d+1}\ p_0 = a_0 \uparrow (s_d\ p_1) = a_0 \uparrow (s_{d+1}\ p_1) = s_{d+2}\ p_0$. Using induction, it follows that for all $n \in \mathbb{N}$: $s_d\ T = s_{d+n}\ T$.
Finally,

$$s\ T = \lim_{n \to \infty} s_n\ T = s_d\ T = s\ Q[\widehat{0}]$$

□

This theorem enables us to translate any dfs-based algorithm for the evaluation of the score of finite game trees to finite game graphs by adding cycle detection. For example, the following algorithm computes the score of a finite game graph $G = (V, E, r, t)$.

Algorithm 6.7

$$\begin{array}{lll} s\ r & = & \textbf{if } \mathit{marked}\ r \rightarrow 0 \\ & & \textbf{elif } r \in F \rightarrow t\ r \\ & & \textbf{elif } r \in V_\updownarrow \rightarrow \mathit{mark}\ r;\ ss := \updownarrow/ \cdot s* \cdot E\ r;\ \mathit{unmark}\ r;\ ss \\ & & \textbf{fi} \end{array}$$

□

Theorem 6.8

Algorithm 6.7 correctly computes the score of a finite game graph G in time $O(|W|)$, where W is the set of nodes of the graph obtained by unraveling the proper semicycles of G.

Proof Apply theorem 6.6 to the tree resulting from unwinding and unraveling G and translate the result back in terms of the original graph. In order to detect cycles, graph marking is used. In every node a bit is maintained indicating whether the node lies on the current path from the root. It is cleared when the graph is generated. It is set (*mark*) just before the descendants of the node are visited, and cleared (*unmark*) just afterwards. Clearly each node in the unraveling of G is visited once. In particular, if the graph contains no proper semicycles, it takes time $O(|V|)$.
□

In the same way other well-known evaluation algorithms, like the α-β algorithm, may be translated to finite game graphs.

7 Memoization

When subtrees are shared (proper semicycles), computation of the score may be accelerated if memoization is used. The score of a subtree is (temporarily) stored and may be looked up later if the same node is reached via a different path (cf. transposition tables in chess programs [Newborn]). Care must be taken when this is applied to the above algorithm, since not every value returned by c for a subtree is a valid s-value!

Example 7.1 Consider a game graph which contains the graph in figure 1 as a subgraph. Suppose the dfs-algorithm visits a first. This triggers the evaluation of c. c is assigned score $0 \downarrow d$, leading to the correct a-score of $b \uparrow (d \downarrow 0)$. If later on c is visited again (by a different path) and its previous score was memoed, the wrong score for c will be produced (the correct value is $d \downarrow (b \uparrow 0)$).
□

However, not all is lost. In the previous example, the score of c was calculated based on the assumption that this value would be used for the calculation of the score of a. In fact, if c is visited later on and a is still on the current path, then the previous score of c can still be used! We'll call the memoed value of score c *depending upon* a. This means that c lies on a cycle which closes at a. In order to accomodate memoization we adapt the algorithm 6.7. We use an additional function c which not only returns the score but also the set of the nodes upon which this score depends.

The following primitives are assumed for doing memoization. The memo values are maintained in the nodes.

- $putmemo : V \times K \times 2^V \to K \times 2^V$.
  $putmemo\ (p, v, d)$ enters v as score of p depending on the set d and also returns (v, d). (2^V denotes the powerset of V).
- $memoed : V \to 2^V_\perp$.
  $memoed\ p$ returns the last value of d which was entered for p using *putmemo*, or $\perp$ if no such value exists. For all lists l: $(\perp \subseteq l) = false$.
- $getmemo : V \to K \times 2^V$.
  $getmemo\ p$ retrieves the last value of (v, d) stored by *putmemo* for p (only used if $memoed\ p \neq \perp$).

Algorithm 7.2

$$
\begin{array}{lcl}
s\ r & = & \pi_1(c\ []\ r) \\
\textbf{where} & & \\
c\ l\ p & = & \textbf{if}\ p \in l \to (0, \{p\}) \\
 & & \textbf{elif}\ (memoed\ p) \subseteq l \to getmemo\ p \\
 & & \textbf{elif}\ p \in F \to putmemo\ (p, tp, \{\}) \\
 & & \textbf{elif}\ p \in V_\updownarrow \to putmemo\ (p, mms, mmd \setminus \{p\}) \\
 & & \quad \textbf{where}\ mms = \updownarrow / \cdot (\pi_1 *) \cdot suc \\
 & & \qquad\qquad mmd = \cup / \cdot (\pi_2 *) \cdot suc \\
 & & \qquad\qquad suc = (c\,(p : l)) * \cdot Ep \\
 & & \textbf{fi}
\end{array}
$$

□

π_1 and π_2 denote projection functions operating on pairs. In order not to distract from the essentials, this algorithm shows a simpler way to detect cycles. The first parameter of c is a list of the nodes on the current path from the root excluding the current node. For efficiency, however, graph marking could be used.

Theorem 7.3

Algorithm 7.2 correctly computes the score of a finite game graph.

Proof

A memoed score may be used if it depends only on nodes on the current path. For $p \in V_?$ the score is computed as before, viz. by taking the maximum/minimum of the subscores in *suc*. The set of nodes on which its score depends is just the union of the dependency sets of the subscores. However, a dependency upon itself may be discharged.

□

The following example shows that this algorithm succeeds in reducing the number of nodes visited from exponential to quadratic in $|V|$.

Example 7.4 See the example in figure 6. To every node there is also a leaf attached. Let L_n be a ladder graph with n layers, apart from the top node. L_n contains a total of $4n+2$ nodes, half of which are leaves. Tests show that the number of nodes visited by the dfs-algorithm without memoization equals $(3n+9)2^{n-1}-2 = O(n2^n)$, whereas with memoization this reduces to $\frac{1}{2}(9n^2-5n+16) = O(n^2)$.

□

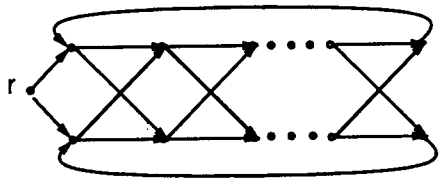

Figure 6: 'Ladder' graph with cycles.

8 Remarks and future work

Motivated by the examples, we conjecture that algorithm 7.2 visits $O(|V|^2)$ nodes. There is room for improvement of the algorithms and for sharpening the upperbounds. For example, it may be that convergence of s only depends on $d_r = \max_{v \in V} d(r,v)$, where $d(a,b)$ is the length of the shortest path from a to b.

Some of the results may be generalized. For example, the requirement that a game graph should be bipartite is not necessary. All that is needed is that V is partitioned into 2 sets V_+ and V_-. This allows for games in which the moves are not strictly alternating. Other directions for extensions include dropping some of the finiteness assumptions (e.g. the finite branching requirement), and allowing K to be a (complete) lattice.

An important and interesting extension concerns the incremental evaluation of the score when part of the graph changes. This applies to the situation where the game graph is only a part of the 'real' game graph, and which changes every time a move is made.

Finally, the connections between this work and the work of [Gijlswijk et al.] could be investigated. They use a completely different, bottom-up algorithm to analyse the L-Game, which runs in time $O(D \times |V|)$, where D is the maximum of the in- and outdegrees of all nodes, and where essentially $K = \{-1, 0, +1\}$.

A Appendix

A.1 Proof of the claim in theorem 6.6

In this section we will prove that $s_d\ T = s_{d+1}\ T$, where T is the unwinding of a simple cycle of length d. This result was used in the proof of theorem 6.6.

Define an M-expression as

$$M\ [a_0, ..., a_{n-1}]\ x = a_0 \updownarrow_0 (a_1 \updownarrow_1 \cdots (a_{n-1} \updownarrow_{n-1} x))$$

where an operator $\updownarrow_j \in \{\uparrow, \downarrow\}$ is associated with each a_j.

We will prove that a recurring occurrence of a value in an M-expression with an identical associated operator may be absorbed.

Theorem A.1 For all $n \geq 2, x \in K, [a_0, \ldots, a_n]$:
If $a_0 = a_n$ and $\updownarrow_0 = \updownarrow_n$, then

$$M\ [a_0, ..., a_n]\ x = M\ [a_0, ..., a_{n-1}]\ x$$

Proof By induction on n.
Base:

$$\begin{array}{lcl} M\ [a_0, a_1]\ x & = & \text{“ def. of } M \text{ ”} \\ a_0 \updownarrow_0 (a_1 \updownarrow_1 x) & = & \text{“}a_0 = a_1, \updownarrow_0 = \updownarrow_1\text{”} \\ a_0 \updownarrow_0 (a_0 \updownarrow_0 x) & = & \text{“}\updownarrow_0 \text{ assoc. and idempotent”} \\ a_0 \updownarrow_0 x & = & \text{“ def. of } M \text{ ”} \\ M\ [a_0]\ x & & \end{array}$$

Step:

$$\begin{array}{lcl} M\ [a_0, ..., a_n]\ x & = & \text{“ def. of } M \text{ ”} \\ a_0 \updownarrow_0 (a_1 \updownarrow_1 \cdots (a_n \updownarrow_n x)) & = & \text{“ } \updownarrow_0 \text{ distributes over } \updownarrow_1 \text{ ”} \\ (a_0 \updownarrow_0 a_1) \updownarrow_1 (a_0 \updownarrow_0 (a_2 \updownarrow_2 \cdots (a_n \updownarrow_n x))) & = & \text{“ def. of } M \text{ ”} \\ (a_0 \updownarrow_0 a_1) \updownarrow_1 (M\ [a_0, a_2, \ldots, a_n]\ x) & = & \text{“ ind. hyp. ”} \\ (a_0 \updownarrow_0 a_1) \updownarrow_1 (M\ [a_0, a_2, \ldots, a_{n-1}]\ x) & = & \text{“ def. of } M \text{ ”} \\ (a_0 \updownarrow_0 a_1) \updownarrow_1 (a_0 \updownarrow_0 (a_2 \updownarrow_2 \cdots (a_{n-1} \updownarrow_{n-1} x))) & = & \text{“ } \updownarrow_0 \text{ distributes over } \updownarrow_1 \text{ ”} \\ a_0 \updownarrow_0 (a_1 \updownarrow_1 \cdots (a_{n-1} \updownarrow_{n-1} x)) & = & \text{“ def. of } M \text{ ”} \\ M\ [a_0, ..., a_{n-1}]\ x & & \end{array}$$

□

Let $T \in \mathbf{GT}_\omega$ be a tree of the following form: T consists of a path $\dot{T} = p_0, \ldots, p_d = \dot{P}$ to the root of a subtree P which is isomorphic to T, and to every p_j $(j \neq d)$ there is a leaf attached with t-value a_j.

Corollary A.2

$$s_d\ T = s_{d+1}\ T$$

Proof

This is a special case of theorem A.1, where

$$\begin{array}{lcl} s_d\ p_0 & = & M\ [a_0, \ldots, a_{d-1}]\ 0 \text{ and (since } P \cong T) \\ s_{d+1}\ p_0 & = & M\ [a_0, \ldots, a_{d-1}, a_0]\ 0 \end{array}$$

and the associated operators alternate, i.e. $\updownarrow_0 = \uparrow$, $\updownarrow_1 = \downarrow$, ... (assuming $p_0 \in V_\uparrow$).

□

Summary of notation

$\|S\|$	cardinality of set S
$f\ a$	function application (left associative)
$/$	reduce operator
$*$	map operator
$\cdot$	apposition operator
$[]$	empty list
$a{:}l$	list with element a appended to list l
$[a,b]$	$= a{:}b{:}[]$
G	finite graphs, see def. 2.1
GG	finite game graphs
G_ω	countable graphs
GG_ω	countable game graphs
T_ω	countable trees
GT_ω	countable game trees
$\cong$	isomorphism of game graphs
$E\ p$	$= \{q \in V \mid (p,q) \in E\}$, where $G = (V,E) \in \mathsf{G}$
$\updownarrow$	used in the context **if** $p \in V_\updownarrow \rightarrow \updownarrow / x$ **fi** = **if** $p \in V_\uparrow \rightarrow \uparrow / x$ **elif** $p \in V_\downarrow \rightarrow \downarrow / x$ **fi**
$\dot{G}$	root of $G \in \mathsf{GG}_\omega$
$P\ T$	P is a subtree of $T \in \mathsf{T}_\omega$
$P\ T$	P is a proper subtree of $T \in \mathsf{T}_\omega$
$T[Q/P]$	substitute $Q \in \mathsf{T}_\omega$ for P in T, where $P\ T \in \mathsf{T}_\omega$
$\hat{n}$	for $n \in K$: the game tree $(\{r\}, \varnothing, r, \lambda r.n)$
$\mathcal{C}_\omega$	class of contexts in GT_ω

References

Elwyn R. Berlekamp, John H. Conway and Richard K. Guy. *Winning Ways for your mathematical plays, Volume 1: Games in General.* Academic Press, 1982.

Richard S. Bird. *Lectures on Constructive Functional Programming.* Technical Monograph PRG-69. Oxford University Computing Laboratory, 1988.

T.B. Boffey. Applying the minimax rule over graphs which are not trees. *Inf. Proc. Letters* 2, 1973, 79–81.

Edward de Bono. *The Five-day Course in Thinking.* Pelican, 1969.

V.W. Gijlswijk, G.A.P. Kindervater, G.J. van Tubergen and J.J.O.O. Wiegerinck. *Computer Analysis of E. de Bono's L-Game.* Report 76-18, Dept. of Mathematics, Univ. of Amsterdam, 1976.

F. Harary. *Graph Theory.* Addison-Wesley, 1969.

Jacques Loeckx and Kurt Siebert. *The Foundations of Program Verification.* Wiley, 1984.

Monroe Newborn. Computer Chess: Ten Years of Significant Progress. In: Marshall C. Yovits (ed.), *Advances in Computers*, Vol. 29, 198–250. Academic Press, 1989.