Fast Algorithms for the TRON Game on Trees

H. Bodlaender and T. Kloks

RUU-CS-90-11
March 1990

Utrecht University

Department of Computer Science

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

Fast Algorithms for the TRON Game on Trees

H. Bodlaender and T. Kloks

Technical Report RUU-CS-90-11
March 1990

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

ISSN:0924-3275

Fast Algorithms for the TRON Game on Trees

H. Bodlaender* T. Kloks†
Department of Computer Science, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands

Abstract

TRON is the following game which can be played on any graph: Two players choose alternately a node of the graph subject to the requirement that each player must choose a node which is adjacent to his previously chosen node and such that every node is chosen only once. In this paper $O(n)$ and $O(n\sqrt{n})$ algorithms are given for deciding whether there is a winning strategy for the first player when TRON is played on a given tree, for the variants with and without specified starting nodes, respectively. The problem is shown to be both NP-hard and co-NP-hard for connected undirected graphs in general.

1 Introduction

Games are not only a popular pastime, but can also serve as a model for several different phenomena, like conflicts between parties with different interests, fault tolerance, worst-case complexity of algorithms (see e.g. [6]), and complexity theory (see e.g. [5]). In this paper we concentrate on the problem to determine whether there is a winning strategy for the first (or second) player in a given game-instance from a certain class of games.

We consider the following type of game: The game is played on a given directed or undirected graph. Two players (called player 1 and player 2) alternately choose a node that has not been chosen before (by either player), and (if it is not his first node) that is adjacent to the last node that has been chosen by that player. In other words, the players are forming two vertex disjoint paths in the graph. The first player that is unable to choose a node loses the game. The game is called TRON, because it can be seen as a generalization of a popular video game with this name. A variant of the game is where the starting positions of the players are specified.

*This author is partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract no. 3075 (project ALCOM)

†This author is supported by the Foundation for Computer Science (S.I.O.N.) of the Netherlands Organization for Scientific Research (N.W.O.).

TRON games were proposed in [2], and the problem to decide whether there is a winning strategy for player 1 for TRON on a directed graph was proven to be PSPACE-complete. The same result was obtained for the variant with specified starting nodes. The complexity of the problem for undirected graphs is not completely determined but as is shown in Section 6, TRON is NP-hard and co-NP-hard for connected, undirected graphs. Therefore, it is useful to look at the complexity of TRON for special classes of graphs. (The complexity of several other games on graphs is considered in [1, 3, 4, 7, 8] among others.)

In this paper we give algorithms to decide whether there is a winning strategy for player 1 on a given tree, both for the version with specified starting nodes and for the version without specified starting nodes. In Section 2 we give a linear time algorithm for the game on trees with specified starting nodes. In Section 3 we give some definitions and some preliminary results for the version without specified starting nodes. We shall show that the problem can be split into two easier problems, of which the outcome decides the outcome of the TRON-problem. In Section 4 we describe an $O(n \log^2 n)$ algorithm to determine one of the subproblems. In Section 5 an $O(n\sqrt{n})$ algorithm is given to decide the other subproblem. In Section 6 we show that TRON on connected, undirected graphs is both NP-hard and co-NP-hard.

2 A linear time algorithm for the version with specified starting nodes

In this section we show the following result:

Theorem 2.1 *There exists a linear time algorithm that, given a tree $T = (V, E)$ and nodes $v_1, v_2 \in V$, determines whether there is a winning strategy for player 1 for* TRON *played on T with starting nodes v_1, v_2.*

Proof:
Let s_1, s_2 be the given start-positions of player 1 and 2. Number the nodes of the path from s_1 to s_2: $0, \ldots, k$. For the first move of player 1, we consider two possibilities. Either he chooses to move towards player 2 (i.e., to position 1), or he decides to diverge from the path. In the latter case player 2 can move freely during the game towards any node of the path except node 0. Let $a^*(i)$ be the length of a maximum path with node i as an endnode and which uses no other nodes (besides i) of the path from s_1 to s_2. Then, at his first move, player 1 will diverge from the path towards player 2 only if:

$$a^*(0) > \max_{1 \leq z \leq k} (k - z + a^*(z))$$

The righthandside is the length of the longest path player 2 can choose if player 1 has diverged from the path at his first move. The computation of the numbers

$a^*(i)$, $i = 0 \ldots, k-1$ clearly takes linear time. Assume that these numbers have been calculated.

Also on later moves, players can either move on the path between s_1 and s_2 or diverge from this path. The next step of the algorithm is the computation of:

$$t_1(i) = \max_{i \le z \le k+1-i}(k+1-i-z+a^*(z)), \quad i = 1, \ldots, \lceil \frac{k}{2} \rceil.$$

$t_1(i)$ is the maximum length of a path that can be made by player 2 after player 1 has diverged at its i^{th} move. Suppose both players have made $i-1$ moves towards each other. Then player 1 will be in position $i-1$, player 2 will be in $k+1-i$, and player 1 must make a move. Player 1 will then diverge from the path only if $a^*(i-1) > t_1(i)$. Similarly, define:

$$t_2(i) = \max_{i \le z < k+1-i}(z-i+a^*(z)), \quad i = 1, \ldots, \lfloor \frac{k}{2} \rfloor.$$

If player 1 is in position i and player 2 is in position $k+1-i$, player 2 has to make the next move. He will decide to leave the path only if $a^*(k+1-i) > t_2(i)$. The calculation of t_1 and t_2 also takes linear time. Now consider the following invariant:

- $0 \le i \le \lfloor \frac{k}{2} \rfloor$
- $\forall_{j:0 \le j < i}(a^*(j) \le t_1(j+1))$
- $\forall_{j:0 \le j < i}(a^*(k-j) \le t_2(j+1))$

This states that none of the players can win the game by diverging from the path before the i^{th} move. Now consider the special case $i = \lfloor \frac{k}{2} \rfloor$. If k is even, player 1 wins the game because, on the one hand, player 2 should move to this position to win the game but, on the other hand, this position is already taken by the first player. If k is odd, player 1 wins the game if and only if $a^*(\lfloor \frac{k}{2} \rfloor) > a^*(\lceil \frac{k}{2} \rceil)$. Thus we find the following algorithm for determing for which player there is a winning strategy for TRON on T with starting nodes s_1, s_2:

$i := 0$;
do $i < \lfloor \frac{k}{2} \rfloor$ and $(a^*(i) \le t_1(i+1) \ \wedge \ a^*(k-i) \le t_2(i+1)) \rightarrow i := i+1$ od;

if $i = \lfloor \frac{k}{2} \rfloor \ \wedge \ k$ even$\rightarrow$ player 1 wins
$\| \ i = \lfloor \frac{k}{2} \rfloor \ \wedge \ k$ odd $\rightarrow$ if $a^*(\lfloor \frac{k}{2} \rfloor) > a^*(\lceil \frac{k}{2} \rceil) \rightarrow$ player 1 wins
$\| \ a^*(\lfloor \frac{k}{2} \rfloor) \le a^*(\lceil \frac{k}{2} \rceil) \rightarrow$ player 2 wins
fi
$\| \ i < \lfloor \frac{k}{2} \rfloor \rightarrow$ if $a^*(i) > t_1(i+1) \rightarrow$ player 1 wins
$\| \ a^*(i) \le t_1(i+1) \rightarrow$ player 2 wins
fi
fi.

□

3 Definitions and preliminary results for the version of TRON without specified starting nodes

Now we consider the TRON game on a tree without specified starting nodes. The *centre* of the tree will play a crucial role in our analysis and algorithms.

Definition 3.1
The *eccentricity* $e(v)$ of a node $v \in V$ in a tree $T = (V, E)$ is the maximum length of a path starting at v. The *radius* $r(T)$ of a tree T is the minimum of the eccentricities of the nodes. A node v is called a *central node* if $e(v) = r(T)$, and the *centre* of T is the set of all central nodes.

It is well-known that the centre of a tree is not empty and always consists of one or two adjacent nodes. The following theorem shows that, although no starting positions are given, for all nodes $v \in V$ but possibly one, player 1 loses when he starts in that node v. This considerably simplifies the problem.

Theorem 3.1
If the centre of the tree consists of two nodes then the second player always has a winning strategy. If the centre of the tree consists of one node then the first player must start in this node, or else the second player will have a winning strategy.

Proof:
Suppose the first player starts in a node s_1, such that there remains an unused central node. Player 2 then chooses the first node on the path from s_1 to a central node. He can then always make a path of at least the same length as player 1, so he wins the game. □

In the rest of this paper it is assumed that the centre of the tree consists of one node and that the first player starts in the central node. We refer to this central node as the *root* of the tree. For future use we also recall the definition of the *centroid* of a tree.

Definition 3.2
A *branch* at a node v is a maximal subtree containing v as a leaf. The *weight* of a node v is the maximum number of edges in any branch at v. A node is called a *centroid-node* if it has minimal weight, and the *centroid* of the tree is the set of all centroid-nodes.

Like the centre, the centroid of a tree either consists of one node or two adjacent nodes. Our algorithm uses a divide and conquer technique, based on the following well-known result.

Lemma 3.1
If the tree has a unique centroid-node, then all the branches have less than $\frac{n}{2}$ *edges,*

where n is the number of nodes in the tree. If the tree has two centroid nodes, then the branch at one of the centroid-nodes, containing the other centroid-node, contains exactly $\frac{n}{2}$ edges.

Definition 3.3
For a node x, let $d(x)$ be the distance of x to the root. We denote by P_x the path from the root to this node. $P_x(i)$, for $i = 0, \ldots, d(x)$ is the i^{th} node of this path, so $P_x(0)$ is the root, $P_x(d(x)) = x$, $P_x(d(x) - 1)$ is the father of x, denoted also by $v(x)$ (if x is not the root), and so on.

Consider the case in which player 2 has chosen a starting node s. Let $k = d(s)$. Now, the players will alternately choose a node. For some time they will move towards each other. Suppose that player 1 is the first to diverge from the path, and that he does so on the i^{th} move. At this point, player 1 will be in position $P_s(i)$ and player 2 is in node $P_s(k - i)$. Let $a_s^*(x)$ be the maximum path-length from node $P_s(d(x))$, which does not use any other node (besides x) of P_s. Then, if player 1 can win the game, he will diverge from the path only if:

$$a_s^*(i) > \max_{i<z\leq k-i}(k - i - z + a_s^*(z))$$

The right-hand side is simply the longest path player 2 can make from his current position. Of course there might be some other node satisfying the equation above, but in choosing the first, player 1 has a winning strategy. If such a node exists we will call it the *winning node* for player 1 (with starting node s for player 2). These considerations lead to the following two subproblems:

Subproblem 1 Given the starting position of player 2, determine the winning node for player 1. This is like playing TRON with the following extra restriction: Player 2 is not allowed to diverge from the path towards the root until player 1 has done so.

Likewise we can define a winning node for player 2, given a starting node s; this is the first node on the path to the root at which player 2 can win the game by diverging from the path (when he is at move).

Subproblem 2 Determine the winning node for player 2 if it exists.

Definition 3.4
For starting node s of player 2, let $r_i[s]$ be the winning node for player i if it exists and let $r_i[s] = \lceil \frac{d(s)}{2} \rceil$ otherwise ($i = (1, 2)$).

The following theorem illustrates that the two subproblems are indeed the only subproblems we have to solve.

Theorem 3.2
Player 2 wins the game by starting at node s if and only if: $r_1[s] > d(s) - r_2[s]$

Proof:
If there is a winning node for one of the players, the player who reaches his winning node first wins the game. So in this case the theorem holds. If there is no winning node for either player, then player 2 wins if and only if $d(s)$ is odd. Then also in this case the theorem holds, since $\lceil \frac{k}{2} \rceil > k - \lceil \frac{k}{2} \rceil = \lfloor \frac{k}{2} \rfloor$ if and only if k is odd. □

In Section 4 we describe an $O(n \log^2 n)$ algorithm to compute $r_2[s]$ for all s, and in Section 5 we describe an $O(n\sqrt{n})$ algorithm to compute $r_1[s]$ for all s. Hence, we have:

Theorem 3.3 *There exists an algorithm that, given a tree $T = (V, E)$, determines in $O(n\sqrt{n})$ time whether there is a winning strategy for player 1 for* TRON *played on T without specified starting nodes.*

4 An $O(n \log^2 n)$ algorithm to determine the winning node of player 2 for every starting position

Suppose player 2 starts at position s at distance k from the root. As before we number the nodes of this path starting at the root $0, \ldots, k$. Notice that if player 2 is at move in position j, player 1 is in position $k - j + 1$. The winning node for player 2 can be defined as follows:

Definition 4.1
$R_2[s] = \{j \mid \lceil \frac{d(s)}{2} \rceil \leq j \leq d(s) \wedge \forall_{z:d(s)-j+1 \leq z < j}(a_s^*(j) > z - (d(s) - j + 1) + a_s^*(z))\}$

Now $r_2[s] = \max_{j \in R_2[s]}(j)$. Notice that for all starting positions s with $k = d(s) \geq 1$: $\lceil \frac{k}{2} \rceil \in R_2[s]$. We can deal efficiently with the numbers $a_s^*(i)$ (for all suitable s and i) by 'shifting the path over one position'.

Definition 4.2
For every node x in the tree except the root, let $a(x)$ be the length of a maximal path with $v(x)$ (the father of x) as an endpoint, which uses no other nodes of P_x besides $v(x)$. Let $l(x)$ be the maximum path-length among paths from x which do not go through $v(x)$. Let $a(root) = l(root) = e(root)$.

Notice that for any starting position s, we have:

$$a_s^*(i) = \begin{cases} a(P_s(i+1)) & i = 0, \ldots, d(s)-1 \\ l(s) & i = d(s) \end{cases}$$

If it is clear which starting node s we are talking about, we also write $a(i)$ in stead of $a(P_s(i))$. Clearly the numbers $a(x)$ and $l(x)$ can be computed in linear time. We can reformulate the definition of r_2 in terms of a and l as follows:

Definition 4.3
$R_2^*[s] = \{j \mid \lceil \frac{d(s)}{2} \rceil \leq j < d(s) \wedge \forall_{z:d(s)-j+1 \leq z < j}(d(s)-j+1+a(j+1) > z+a(z+1))\}$

Note that

$$r_2[s] = \begin{cases} k & \text{if } k = 1 \text{ or } \forall_{1 \leq z < k}(1 + l(k) > (z + a(z+1)) \\ \max\limits_{j \in R_2^*[s]}(j) & \text{otherwise} \end{cases}$$

We use the following notation: $r_2^*[s] = \max_{j \in R_2^*[s]}(j)$. These formulas may look alarming at first sight, but when looking at the term $k - j + 1$ in $R_2^*[s]$ we notice some monotony. So we replace the term $k - j + 1$ by a new variable.

Definition 4.4
For any node x in the tree we define: $(k = d(x))$

$$\begin{aligned} \Delta(x) &= \{s \mid 0 \leq s < k \wedge \forall_{z:s \leq z < k-1}(a(k) > z - s + a(z+1))\} \\ \delta(x) &= \min_{s \in \Delta(x)}(s) \end{aligned}$$

Lemma 4.1 *Let $k = d(s)$. Then $R_2^*[s] = \{j \mid \lceil \frac{k}{2} \rceil \leq j < k \wedge \delta(P_s(j+1)) \leq k - j + 1\}$.*

Proof:
This is the monotony we were talking about: $\delta(P_s(j+1)) \leq q$ if and only if $\forall z$: $q \leq z < j(q + a(j+1) > z + a(z+1))$. □

It is straightforward to determine in linear time the nodes s for which $r_2[s] = d(s)$. It remains to compute $r_2^*[x]$, for all $x \in V$. The computation of these numbers uses two phases. In the first phase we determine the numbers $\delta(x)$ and in the second phase we compute $r_2^*[x]$. The algorithms use divide-and-conquer techniques.

4.1 The computation of $\delta(x)$

Consider a subtree with a root w which has m nodes. We describe a subroutine which computes the numbers $\delta_w(x)$ for all nodes in the subtree except w:

$$\begin{aligned} \Delta_w(x) &= \{s \mid d(w) \leq s < k \ \wedge \ \forall_{z:s \leq z < k-1}(a(k) > z - s + a(z+1))\} \\ \delta_w(x) &= \min_{s \in \Delta_w(x)}(s) \end{aligned}$$

In this formula $k = d(x)$ is the distance to the *original* root.

The case $m \leq 2$ takes constant time. Now assume $m > 2$. The first step of this algorithm is the computation of the centroid-node c of the subtree which is farthest away from w. This can be done in linear time. The second step is the recursive call of the subroutine for each son of c (or more precise: for the subtree with that son as a root). Next we delete all subtrees of c and we recursively call the subroutine for the adjusted subtree rooted at w (which now has c as a leaf). Notice that for nodes in this last subtree δ_w has been determined correctly. For nodes in the subtree at c we split the path to w. So the third step is the computation of numbers $m_w(x)$ for all nodes in the subtree rooted at c:

$$m_w(x) = \max_{d(c) \leq z < d(x)-1}(z + a(z+1))$$

and the computation of numbers $\gamma(s)$ for $d(w) \leq s < d(c)$:

$$\gamma(s) = \max_{s \leq z < d(c)}(z + a(z+1)),$$

and we define $\gamma(d(c)) = 0$. This step clearly takes at most linear time.

The fourth step is the computation of δ_w for the sons of c. If x is a son of c, we have:

$$\begin{aligned} \Delta_w(x) &= \{s \mid d(w) \leq s \leq d(c) \ \wedge \ s + a(x) > \gamma(s)\} \\ \delta_w(x) &= \min_{s \in \Delta_w(x)}(s) \end{aligned}$$

Since γ is a monotonic non-increasing function, this step takes at most $O(\log m)$ time for each son of c. The fifth step is the computation of $\delta_w(x)$ for the other nodes in the subtree at c. For these nodes we have:

$$\begin{aligned} \tilde{\Delta}_w(x) &= \{s \mid d(w) \leq s \leq d(c) \ \wedge \ s + a(x) > \gamma(s) \ \wedge \ s + a(x) > m_w(x)\} \\ \delta_w(x) &= \begin{cases} \delta_c(x) & \text{if } \delta_c(x) > d(c) + 1 \\ \min\limits_{s \in \tilde{\Delta}_w(x)}(s) & \text{otherwise} \end{cases} \end{aligned}$$

Hence, also this step takes at most $O(\log m)$ time for each node x.

Let $T(m)$ be the time needed for the computation of δ. Since c is the centroid node which is furthest away from w, all subtrees at c contain at most $\lfloor \frac{m}{2} \rfloor$ nodes.

The adjusted subtree rooted at w in which c is a leaf contains at most $\lfloor \frac{m}{2}+1 \rfloor$ nodes. If f_i is the number of nodes in the i^{th} subtree, we find:

$$\begin{aligned} F &= \{(f_1, f_2, \ldots) \mid f_1 + f_2 + \ldots = m \ \wedge \ \forall_i (f_i \leq \lfloor \frac{m}{2} + 1 \rfloor)\} \\ T(m) &\leq \max_{f \in F} (\sum_i T(f_i)) + O(m \log m) \end{aligned}$$

With induction on the number of subtrees at c the following is easy to prove:

$$T(m) \leq T(\lfloor \frac{m}{2} + 1 \rfloor) + T(\lceil \frac{m}{2} - 1 \rceil) + O(m \log m)$$

and finally this gives: $T(m) = O(m \log^2 m)$.

4.2 The computation of $r_2^*[x]$

As stated before we have for nodes x at distance $k = d(x) > 1$ from the root:

$$\begin{aligned} R_2^*[x] &= \{j \mid \lceil \frac{k}{2} \rceil \leq j < k \ \wedge \ k - j + 1 \geq \delta(j+1)\} \\ r_2^*[x] &= \max_{j \in R_2^*[x]} (j) \end{aligned}$$

Let w be a node of the tree T. We describe a subroutine which determines $r_w^*[x]$ for all x in the subtree at w ($k = d(x)$):

$$\begin{aligned} R_w^*[x] &= \{j \mid d(w) \leq j < k \ \wedge \ k - j + 1 \geq \delta(j+1)\} \\ r_w^*[x] &= \begin{cases} \max\limits_{j \in R_w^*[x]} (j) & \text{if } R_w^*[x] \neq \emptyset \\ -1 & \text{otherwise} \end{cases} \end{aligned}$$

Then, for nodes at distance at least 2 from the root we have $r_{root}^*(x) = r_2^*(x)$. In the first step of the algorithm we again compute the centroid-node c of this subtree which is farthest away from w. The second step is the recursive call of the subroutine for the same subtrees as described in the previous section.

Let y be a son of c, and let x be a node in the subtree of y. We first check whether $r_w^*(x) = d(c)$. We have: $r_w^*(x) = d(c)$ if and only if: $r_c^*(x) = -1 \ \wedge \ \delta(y) + d(c) \leq d(x) + 1$. For convenience, we define for these nodes x: $r_c^*(x) = d(c)$. To assign $r_w^*(x)$ for the rest of the nodes in the subtree at c (except c), we first define:

Definition 4.5
For $k = d(c) + 1, \ldots, n$:

$$\begin{aligned} T[k] &= \{j \mid d(w) \leq j < d(c) \ \wedge \ j + \delta(j+1) \leq k + 1\} \\ t[k] &= \begin{cases} -1 & \text{if } T[k] = \emptyset \\ \max\limits_{j \in T[k]} (j) & \text{otherwise} \end{cases} \end{aligned}$$

We can assign the values to array t in $O(n)$ time as follows:
Take the following invariant:

- $k \leq n \ \wedge \ \forall_{l:l>k \wedge d(c)<l\leq n} : t[l] = \begin{cases} -1 & \text{if } T[l] = \emptyset \\ \max\limits_{j \in T[l]}(j) & \text{otherwise} \end{cases}$
- $i < d(c) \ \wedge \ \forall_{j:j>i}(d(w) \leq j < d(c) \Rightarrow j + \delta(j+1) > k+1)$

Then the algorithm becomes:

$k := n$; $i := d(c) - 1$;
do $i + \delta(i+1) > k+1 \ \wedge \ i \geq d(w) \ \rightarrow i := i - 1$
$\|$ $i + \delta(i+1) \leq k+1 \ \wedge \ k > d(c) \rightarrow t : [k] = i$; $k := k - 1$
od;
do $k > d(c) \rightarrow t : [k] = -1$; $k := k - 1$ od

Now, for any node in the subtree of c we can now compute $\tilde{r}^*_w[x] = \max(\tilde{r}^*_c[x], t[d(x)])$. We conclude that this part of the algorithm takes $O(m \log m)$ time.

5 An $O(n\sqrt{n})$ algorithm to determine the winning node for player 1

Suppose player 2 starts at position s at distance k from the root ($k \geq 1$). If player 1 is at move in position i, then player 2 is in position $k - i$. The winning node for player 1 can be defined in terms of a and l as:

Definition 5.1
Let $d(s) \geq 1$. $R^*_1[s] = \{i \mid 0 < i \leq \lceil \frac{d(s)}{2} \rceil \ \wedge \ \forall_{z:i<z\leq d(s)-i}(a(i+1) > d(s) - i - z + a(z+1))\}$.

Now

$$r_1[s] = \begin{cases} 0 & \text{if } a(0) > l(s) \ \wedge \ \forall_{z:0<z<d(s)}(a(0) > d(s) - z + a(z+1)) \\ \min\limits_{i \in R^*_1[s]}(i) & \text{otherwise} \end{cases}$$

As a shorthand notation we use: $r^*_1[s] = \min_{i \in R^*_1[s]}(i)$. Clearly, also in this case we only have to concentrate on the computation of $r^*_1[s]$. In this case we decide to replace the term $d(s) - i$ in R^*_1 by a new variable:

Definition 5.2
$S(x) = \{i \mid 0 < i \leq d(x) \ \wedge \ \forall_{z:i<z<d(x)}(a(i+1) \geq d(x) - z + a(z+1))\}$

Now we want to look at the maximum position player 2 can be in such that player 1 can still win the game by diverging from the path in position i.

Definition 5.3
For $i = 1, \ldots, d(s)$ we define $(k = d(s))$:

$$\begin{aligned} \Delta_s(i) &= \{j \mid i \leq j \leq k \ \wedge \ i \in S(j)\} \\ \delta_s(i) &= \max_{j \in \Delta_s(i)}(j) \end{aligned}$$

Lemma 5.1 *If* $\tilde{R}_1[s] = \{0 < i \leq d(s) \ \wedge \ i + \delta_s(i) > d(s)\}$, *then* $r_1^*[s] = \min_{i \in \tilde{R}_1[s]}(i)$.

Proof:
First notice that for $i > 0$ we have: $(k = d(s))$

$$i + \delta_s(i) > k \Leftrightarrow \forall_{z:i<z\leq k-i}(a(i+1) > k - i - z + a(z+1))$$

Also, for $0 < i < k$: $\delta_s[i] \geq i + 1$. This implies: $\exists_{i:0<i\leq\lceil\frac{k}{2}\rceil}(i + \delta_s(i) > k)$ (and this also holds for $k = 1$). This proves the lemma. □

5.1 The computation of the (ordered) sets $S(x)$

In this section we describe an $O(n\sqrt{n})$ algorithm to determine the set $S(x)$ for every node x:

$$S(x) = \{i \mid 0 < i \leq d(x) \ \wedge \ \forall_{z:i<z<d(x)}(a(P_x(i+1)) \geq d(x) - z + a(P_x(z+1)))\}$$

So, for example we have defined $S(root) = \emptyset$; if $d(x) > 0$ then $d(x) \in S(x)$; and if $d(x) > 1$ then we have $(d(x) - 1) \in S(x)$. First we shall show that the only candidates for $S(x)$ besides $d(x)$ are in $S(v(x))$ (when x is not the root).

Lemma 5.2 $0 < i < d(x) \ \wedge \ i \in S(x) \Rightarrow i \in S(v(x))$

Proof:
We have, if $0 < i < d(x)$:

$$\begin{aligned} & \forall_{z:i<z<d(x)}(a(P_x(i+1)) \geq d(x) - z + a(P_x(z+1))) \\ \Rightarrow \ & \forall_{z:i<z<d(x)-1}(a(P_x(i+1)) \geq (d(x) - 1) - z + a(P_x(z+1))) \\ \Rightarrow \ & \forall_{z:i<z<d(x)-1}(a(P_{v(x)}(i+1)) \geq (d(x) - 1) - z + a(P_{v(x)}(z+1))) \\ \Rightarrow \ & i \in S(v(x)) \end{aligned}$$

□

Consider a node w. We describe a procedure which determines $S(x)$ for every node x in the subtree of w. We keep the following invariant:

- w is a node in the tree.
- $S(w)$ has been determined.
- For all $i \in S(w)$ with $i < d(w) - 1$: $\omega(i) = \max_{i<z<d(w)}(a(P_w(z+1)) - z)$
- For all $i < d(w)$: $a^*(i) = a(P_w(i+1))$

```
proc determine-S-in-subtree(w);
    for x ∈ sons of w do
        S : (x) = ∅; k := d(x); a* : (k − 1) = a(x);
        for i ∈ S(w) do
            if i < d(w) − 1 →
              Ω : (i) = ω(i);
              ω : (i) = max(ω(i), a(x) − k + 1)
            ‖ i = d(w) − 1 → ω : (i) = a(x) − k + 1
            ‖ i ≥ d(w) → skip
            fi;
            if i ≤ d(w) − 1 and a*(i) ≥ k + ω(i) → put i in S(x)
            ‖ i > d(w) − 1 or a*(i) < k + ω(i) → skip
            fi
        rof;
        put k − 1 and k in S(x);
        determine-S-in-subtree(x);
        for i ∈ S(w) do
            if i < d(w) − 1 → ω : (i) = Ω(i)
            ‖ i ≥ d(w) − 1 → skip
            fi
        rof
    rof
corp
```

We assume the sets are represented as an array-like structure. (This allows us to perform binary search on the sets.) For every node x the amount of time needed to compute $S(x)$ is clearly bounded by a constant times $\mid S(v(x)) \mid$ (if x is not the root). By the following lemma, it now follows that the time needed by the algorithm of this subsection is $O(n\sqrt{n})$.

Lemma 5.3
For every node y: $\mid S(y) \mid \leq \sqrt{2n} + 2$.

Proof:
Consider two consecutive elements of $s(y)$, say i and j. Let $i < j \leq d(x)$. Then, if $j < d(x)$, by definition:

$$a(P_y(i+1)) \geq d(x) - j + a(P_y(j+1)) > a(P_y(j+1))$$

So the a-values of $S(y)$ form a decreasing subsequence of P_y (except for the last element of $S(y)$). But as these values correspond to the lengths of disjoint paths, we have that $\sum_{t=1}^{d(y)} a(P_y(t)) + l(y) \leq n$. So $\sum_{t \in S(y) \setminus d(y)} a(P_y(t+1)) \leq n$. But the longest monotonic sequence of nonnegative integers with sum less than or equal to n contains at most $1 + \sqrt{2n}$ elements. This proves the lemma. □

5.2 The computation of $r_1^*[s]$

In this subsection we compute the winning node for player 1, for every starting position of player 2, again by using a divide-and-conquer strategy. Consider a node w, and let the number of nodes in the subtree with w as a root, be m. If we want to determine the winning node for player 1 for starting nodes in the subtree with root w, we cannot look only at nodes in the subtree at w, because these winning nodes might be on P_w. If however such a winning node appears in some set $S(v)$ of a node v in the subtree, we can take it into account. We describe a subroutine which determines the winning node for player 1, for every starting position in the subtree of w, based on nodes which appear in some set of a node in the subtree of w. More precisely, we determine for every node x in the subtree of w, except for w, the number $\tilde{r}_w[x]$ defined as follows:

Definition 5.4

$$\begin{aligned} \tilde{R}_w[x] &= \{i \mid 0 < i \leq d(x) \ \wedge \ i + \delta_x(i) > d(x) \ \wedge \ \delta_x(i) \geq d(w)\} \\ \tilde{r}_w[x] &= \min_{i \in \tilde{R}_w[x]} (i) \end{aligned}$$

We must compute $\tilde{r}_{root}[x]$, for all x, since $\tilde{r}_{root}[x] = r_1^*[x]$. We do this by divide-and conquer. When we want to compute $\tilde{r}_w[x]$, for all x in a subtree with root w, we start by determining the centroid-node of this subtree, that is furthest away from w. For every son of c we recursively call the subroutine, for the subtree rooted at this son. Next we delete all sons of c, and we call the subroutine for the adjusted subtree rooted at w, which now has c as a leaf. For nodes in this last subtree the number $\tilde{r}_w$ has been correctly determined. Now consider a node x in the subtree rooted at c. If $\tilde{r}_w[x]$ has not been determined correctly, then there must be a node on P_c, say i, which does not appear in a set of the subtree, and which satisfies:

$$0 < i \leq d(c) \ \wedge \ i + \delta_c(i) > d(x) \ \wedge \ d(c) \geq \delta_c(i) \geq d(w)$$

Notice that these nodes i either must appear in $S(w)$ or on the path from w to c. For these nodes we define:

Definition 5.5
Let $V = S(w) \cup \{j \mid d(w) \leq j \leq d(c)\}$. Let $i \in V$. We define:

$$\begin{aligned} Q[i] &= \{j \mid (j \in V \wedge i \in S(P_c(j))\} \\ q[i] &= \max_{j \in Q[i]}(j) \end{aligned}$$

Since the number of elements in each set is bounded by a constant times $\sqrt{n}$, and since these sets are ordered, the determination of all $q[i]$ takes $O(\sqrt{n} \log m \log \sqrt{n}) + O(m \log m \log \sqrt{n})$ time (in which it is assumed that we can also perform a binary search on the nodes of the path from w to c).

Definition 5.6
For $k = d(c) + 1, \ldots, n$ we define:

$$\begin{aligned} T[k] &= \{j \mid j \in V \wedge j + q(j) > k\} \\ t[k] &= \begin{cases} k & \text{if } T[k] = \emptyset \\ \min\limits_{j \in T[k]}(j) & \text{otherwise} \end{cases} \end{aligned}$$

We can assign the values to array t in $O(n)$ time as follows:
Take the following invariant:

- $d(c) + 1 \leq k \wedge \forall_{l: l < k \wedge l \leq n} : t[l] = \begin{cases} l & \text{if } T[l] = \emptyset \\ \min\limits_{j \in T[l]}(j) & \text{otherwise} \end{cases}$
- $i \in V \wedge \forall_{j: j < i}(j \in V \Rightarrow j + q[j] \leq k)$

Then the algorithm becomes:

$k := d(c) + 1$; $i :=$ first element of V;
do $i + q[i] \leq k \wedge V \neq \emptyset \rightarrow$ take next i
$\|\ \ i + q[i] > k \wedge k \leq n \rightarrow t : [k] = i$; $k := k + 1$
od;
do $k \leq n \rightarrow t : [k] = k$; $k := k + 1$ od

For the sons of c we first set $\tilde{r}_c[son] = d(c)$. Now, for any node in the subtree of c we can now compute $\tilde{r}_w[x] = \min(\tilde{r}_c[x], t[d(x)])$.
Let $T'(m)$ be the time needed by this algorithm. Then we find:

$$T'(m) \leq T'(\lfloor \frac{m}{2} + 1 \rfloor) + T'(\lceil \frac{m}{2} - 1 \rceil) + O(\sqrt{n} \log n \log m) + O(m \log m \log n) + O(n)$$

and this gives: $T'(n) = O(n \log^3 n)$.

6 Complexity of TRON on undirected graphs

In this section we investigate the complexity of TRON on undirected graphs. Although we conjecture that this problem is PSPACE-complete, we were not able to prove this. However, it is fairly easy to show that the problem is both NP-hard and co-NP-hard for connected, undirected graphs.

Theorem 6.1 *The problem to decide whether for a connected, undirected graph $G = (V, E)$ there is a winning strategy for player 1 for* TRON*, played on G without specified starting nodes, is NP-hard.*

Proof:
We use a transformation from HAMILTONIAN PATH with one specified endpoint of the path. (It is easy to see that this problem is NP-complete.) Let an undirected graph $G = (V, E)$ and a node $s \in V$ be given. Let $n = |V|$. (We assume $n \geq 4$.) Let $G' = (V', E')$ be the graph, defined by $V' = V \cup \{w_1, w_2, \cdots, w_{2n-3}\}$, and $E' = E \cup \{(w_i, w_{i+1}) \mid 1 \leq i \leq 2n-4\} \cup \{(s, w_i) \mid 1 \leq i \leq 2n-3\}$. We now claim that player 1 has a winning strategy for TRON on G' without specified starting nodes, if and only if G has a Hamiltonian path, starting with node s. From this claim, the theorem follows by observing that G' can be constructed in polynomial time.

First we remark that player 1 must start in s. If he does not, then player 2 starts in s. If player 1 started in a node $v \in V$, then player 2 wins by moving to w_1: player 1 can make at most $n-1$ moves, and player 2 can make $2 \cdot n - 4$ moves. If player 1 started in a node w_i, and moves in his second move to w_{i-1} (w_{i+1}), then player 2 moves to w_{i-2} (w_{i+2}) and wins directly. So suppose player 1 starts in s.

Player 2 now must start in w_{n-1}. If he instead starts in a node $v \in V$, then player 1 wins by moving to w_1. If he starts in a node w_i, with $i \neq n-1$, then player 1 wins by moving to w_{i+1}, if $i < n-1$, and to w_{i-1} otherwise.

Now player 1 can make $n-1$ moves, if and only if G has a Hamiltonian path starting with node s. Player 2 can always make exactly $n-2$ moves, so wins if this Hamiltonian path does not exist. □

Theorem 6.2 *The problem to decide whether for a connected, undirected graph $G = (V, E)$ there is a winning strategy for player 1 for* TRON*, played on G without specified starting nodes, is co-NP-hard.*

Proof:
Note that HAMILTONIAN PATH is NP-complete, even for graphs that have at least one node with degree 1 and an odd number of nodes. Let $G = (V, E)$ be a graph with $v \in V$ a node with degree 1, and $n = |V|$ odd. Let G' be obtained by taking the disjoint union of G and a complete graph with n nodes, and then adding an

edge between v and one of the nodes x of the complete subgraph. Now player 2 has a winning strategy for TRON on G', if and only if G has a Hamiltonian path.

If there does not exists a Hamiltonian path in G, then player 1 wins by starting in x. If player 2 starts in G, he loses because he can make fewer moves than player 1, and if he starts in the complete subgraph, he loses because n is odd.

Suppose G has a Hamiltonian path. If player 1 starts in G, then player 2 starts in x. If player 1 starts in the complete subgraph, then player 2 starts in v. In both cases, player 2 can make at least the same number of moves as player 1 and hence wins the game. □

Corollary 6.1 *The problem to decide whether for a connected, undirected graph $G = (V, E)$ and nodes $s_1, s_2 \in V$, there is a winning strategy for player 1 for* TRON, *played on G with starting nodes s_1, s_2, is NP-hard.*

Proof:
Use the construction of Theorem 6.1. Take $s_1 = s$, $s_2 = w_{n-1}$. □

Corollary 6.2 *The problem to decide whether for a connected, undirected graph $G = (V, E)$, and nodes $s_1, s_2 \in V$, there is a winning strategy for player 1 for* TRON, *played on G with starting nodes s_1, s_2, is co-NP-hard.*

References

[1] A. Adachi, S. Iwata, and T. Kasai. Some combinatorial game problems require $\Omega(n^k)$ time. *J. ACM*, 31:361–376, 1984.

[2] H. L. Bodlaender. Complexity of path forming games. Technical Report RUU-CS-89-29, Department of Computer Science, University of Utrecht, 1989.

[3] H. L. Bodlaender. On the complexity of some coloring games. Technical Report RUU-CS-89-27, Department of Computer Science, University of Utrecht, 1989.

[4] A. Fraenkel and E. Goldschmidt. Pspace-hardness of some combinatorial games. *J. Combinatorial Theory ser. A*, 46:21–38, 1987.

[5] D. Johnson. The NP-completeness column: An ongoing guide. *J. Algorithms*, 4:397–411, 1983.

[6] K. Mehlhorn, S. Näher, and M. Rauch. On the complexity of a game related to the dictionary problem. In *Proceedings 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 546–548, 1989.

[7] T. J. Schaefer. On the complexity of some two-person perfect-information games. *J. Comp. Syst. Sc.*, 16:185–225, 1978.

[8] L. J. Stockmeyer and A. K. Chandra. Provably difficult combinatorial games. *SIAM J. Comput.*, 8:151–174, 1979.