Euler Circle Paper Combinatorial Games

Victor Donchenko

August 2021

Background on Computation Complexity Theory

In order to establish a model for complexity of computation we must first establish a model for computation in general. Mathematicians often work with computational problems in the form of "decision problems", where the answer is "yes" or "no" for a given finite input string of characters, because this construction is simple and can be used to recreate other notions of computation problems. An example is whether a given string consists of a single string repeated twice. Finite-information structures such as numbers and graphs can be encoded as strings, allowing problems to be constructed utilizing those notions. Thus another decision problem could be whether an integer-weighted graph has a minimum spanning tree of weight at most k for some given integer k. Games can also be encoded in a variety of ways depending on the application and the class of games.

These problems are often represented as the set of strings for which the answer is "yes", which is called a "language". A computation process which produces the answer for a given string is said to "decide" the language. The first example of a type of computation process is the finite state automaton. This computation process has a finite number of "states", including a start state and an accept state. There is a transition function which given the current state and the next character, produces the next state. The automaton begins in the start state and consumes characters one by one. Iff at the last character it is in the accept state then the answer is "yes".

This process type is simple and interesting but really it can decide only a very limited set of languages. It can decide whether a string of as and bs has three as in a row, but it cannot decide whether a string is a string repeated twice, for example. A more sophisticated process type is the Turing machine. Like the finite state automaton, it has a finite number of states and a transition function to move between them, but it also has a "tape" of characters which assist in the state transitions. At the beginning of the computation, the tape contains the input string and the Turing machine's head is placed at the first character of the tape. The Turing machine is also placed in the start state. At every step the Turing machine advances to another state, which is a function of the current state and the character under the head of the machine. This transition also includes writing a new (or the same) character to the previous position of the head and moving the head either left or right. These actions are described by the transition function. If the Turing machine moves into an “accept” or “reject” state the answer is “yes” or “no” respectively.

Now to consider the complexity of a Turing machine process and by extension its language. The Turing machine is a natural candidate for this analysis since it gives rise to immediate and intuitive concepts of complexity and is conceptually rather similar to computers used in practice. There are several notions of the complexity of a Turing machine process, but the most common are “time” and “space”, which refer to how many steps the Turing machine takes and how large the tape gets during the operation, both considered as a function of the size of the input string. A Turing machine is said to be in “polynomial-time” if the number of steps it takes is bounded above by a polynomial in the size of the input string, and similarly for “polynomial-space”. Differences between the Turing machine model and practical computers in memory access disappear when considering polynomial-time and polynomial-space differences to be inconsequential.

The problem classes referred to by P and PSPACE are the sets of problems decided by Turing machines in polynomial time and polynomial space, respectively. NP is the class of problems which can be verified in polynomial time – that is, those with a Turing machine which can decide in polynomial time whether a given solution (often called a “certificate”) is valid to a given query for the problem. (The name NP comes from an equivalent formulation of it as the set of problems decided in polynomial time by a variation on the process model known as nondeterministic Turing machines.)

A problem in NP for which a polynomial-time Turing machine would give a polynomial-time solution for everything in NP is said to be NP-complete. The first problem to be proven to be NP-complete was SAT, which asks for a given expression of boolean variables involving basic boolean operations, whether there is a consistent assignment of values to the variables such that the expression evaluates to true. Future problems were shown to be able to be reduced to from a previously proven NP-complete problem.

Redwood furniture being NP-complete to evaluate

Recall that redwood furniture denotes a Hackenbush position where all blue edges touch the ground and these are all the blue edges and every blue edge is adjacent to exactyl one red edge. Additionally the position graph is a connected graph. The number for any redwood furniture position is $\frac{1}{2^n}$ for some nonnegative integer n. Finding this n is NP-complete. This is proven by reducing the Steiner tree problem to this problem. The Steiner tree problem asks for an undirected graph G with nonnegative weights and a subset S of its vertices, what is the smallest weight subtree of G which contains S?

Theorem. *For any redwood furniture position G, we have $G = \{0 \mid G^R\}$.*

Proof. Any left move is reversible here. Let G^L be a left move removing a blue edge. Then suppose right responds by removing the red edge adjacent to that edge. Let this be G^{LR}. Consider $G - G^{LR}$. Note that $-G^{LR}$ has fewer grounded feet than G. We wish to show that this is ≥ 0, so suppose right moves first. Left can mirror any move in either component in the other component. After this occurs G will have feet (which would be blue) left over and $-G^{LR}$ would have been toppled, which is an L position, so left wins. Therefore G^{LR} is a reversing move, which is furthermore to a redwood furniture position. Thus this and future left moves can be bypassed repeatedly to obtain $\{0 \mid G^R\}$. Therefore $G = \{0 \mid G^R\}$. □

Theorem. *The number for any redwood furniture position G where the red edges considered by themselves is a tree and for which every red edge touches a blue edge (which we shall call a redwood tree) is $\frac{1}{2}$.*

Proof. We use induction on a, the number of blue edges.

Base case: $a = 1$

In this case the position is the tower BR, which is known to have $\frac{1}{2}$ as its number.

Inductive step: $a > 1$

We assume the claim for smaller a.

If right moves, then the position breaks into two redwood furniture positions with the properties in the claim, with fewer numbers of blue edges. Thus by the inductive hypothesis we have that each is $\frac{1}{2}$, and so G^R is their sum which is 1. We have that $G = \{0 \mid G^R\}$ by an above theorem so $G = \{0 \mid 1\} = \frac{1}{2}$. □

Theorem. *Every redwood furniture position has number $\frac{1}{2^n}$ for some nonnegative integer n.*

Proof. Let G be a redwood furniture position. We have $G = \{0 \mid G^R\}$. We have that G^R is some dyadic rational $\frac{a}{2^k}$. Let the largest power of 2 less than a be 2^b. Then $\frac{2^b}{2^k} = \frac{1}{2^{k-b}}$ is between 0 and G^R. There is no other dyadic rational between 0 and $\frac{a}{2^k}$ with smaller denominator like say 2^c, because for any c less than $k - b$ we have $\frac{1}{2^c}$ is at least $\frac{2^{b+1}}{2^k}$ which is greater than $\frac{a}{2^k}$ by construction of b. Therefore $G = \frac{1}{2^{k-b}}$. □

Note that in general right prefers to keep the blue edges connected since otherwise further on there would be two disconnected redwood tree positions which have a combined number of 1 together.

Theorem. *Every redwood furniture position G which is not a redwood tree is equal to $\frac{1}{2}G^R$.*

Proof. Since right prefers to keep the blue edges connected, G^R is still a redwood furniture position, so it is equal to $\frac{1}{2^n}$ for some nonnegative integer n. Therefore $G = \{0 \mid \frac{1}{2^n}\}$. Thus $G = \frac{1}{2^{n+1}} = \frac{1}{2}G^R$. □

Consider the Steiner problem with a graph G and a set of its vertices G. Take the graph G, make it out of red edges where an edge of weight n is replaced with n edges in a row, and put each vertex in S on a blue foot. Right wants to maximize the number of edges they remove until the position becomes a redwood tree. If k is the maximum number of edges that right can remove before the position becomes such, then the number of the redwood furniture position is $\frac{1}{2^{k+1}}$, since for any redwood furniture position K that is not a redwood tree we have $K = \{0 \mid K^R\} = \frac{1}{2}K^R$. Right wants to keep the vertices corresponding to elements of S connected as per what was previously written. Therefore after right has removed k edges the remaining position is a subtree of the initial position which contains all the vertices corresponding to S and which has the smallest size which is k less than the size of the original position. Since the size corresponds to the total weight of a corresponding subtree of G, this corresponding subtree is the smallest total weight subtree of G spanning S. Calculating k gives the total weight of this subtree, so this is a reduction from the Steiner tree problem to redwood furniture in polynomial time. Thus redwood furniture is NP-complete.