Divide and Conquer

Rich Schwartz

February 9, 2016

1 Dyadic Cubes

A *cube* in $\boldsymbol{R}^n$ is defined to be the product $I_1 \times ... \times I_n$, where the intervals $I_1, ..., I_n$ all have the same length. The cube is a *unit integer cube* if the endpoints of all the intervals are integers and the side length is 1. In other words, the vertices have integer coordinates and the side length is 1.

The *dyadic subdivision* of the integer cube $I_1 \times ... \times I_n$ is the list of 2^n smaller cubes of the form $I_1' \times ... \times I_n'$, where I_k' is either the left half or the right half of I_j for each j. In other words, the dyadic subdivision of a cube is just the list of 2^n cubes you get by cutting the cube in half in all directions. We write $Q_1 \to Q_2$ is Q_2 is one of the cubes in the dyadic subdivision of Q_1.

A *dyadic cube* is any cube Q' obtained by recursively subdividing a unit integer cube. That is, we have $Q = Q_0 \to Q_1 \to ... \to Q_m = Q'$ where Q is a unit integer cube. Here is another quick definition of a dyadic cube. A cube $Q \subset \boldsymbol{R}^n$ is dyadic if and only if there is some positive integer N such that the map $x \to 2^N(x)$ maps Q to a unit integer cube.

It is easiest to think about dyadic cubes contained in the unit cube $[0,1]^n$. The set of dyadic cubes in $[0,1]^n$ has the structure of a directed infinite tree of valence $2^n + 1$. The (directed) edge relation is given by $Q_1 \to Q_2$. Figure 1 shows the case when $n = 1$.

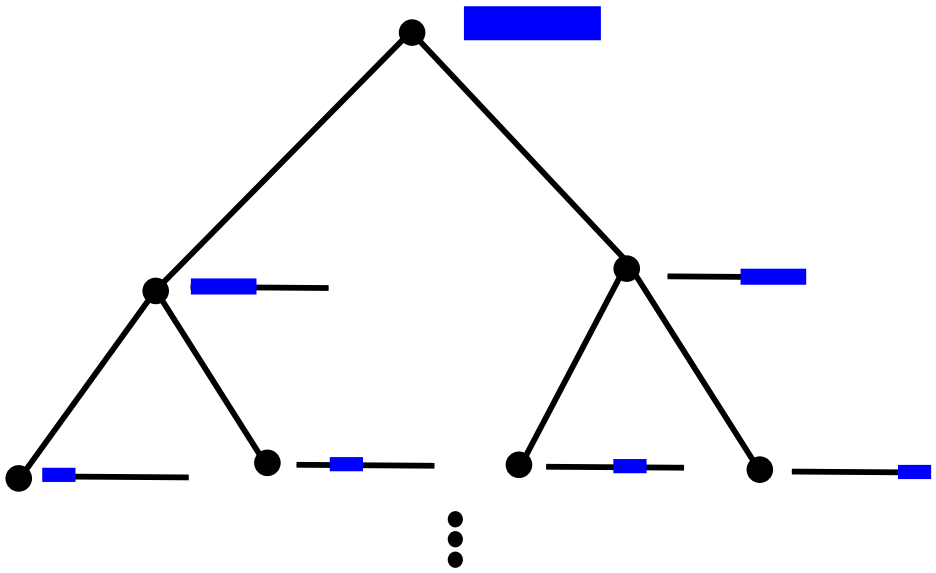

Figure 1: The tree of dyadic intervals

2 Divide and Conquer

Here I'll describe a general kind of algorithm called *divide and conquer*. Suppose you have some property P that you want to check is true on the unit cube $[0,1]^n$. For instance, P might be the statement that some function is positive on $[0,1]^n$. What you want from P is that P is true on the union $A \cup B$ of two sets A and B provided that P is true on A and B separately. With this kind of property, you can prove that P is true on the whole cube by breaking the whole cube into smaller pieces and checking it on each piece. Here is how the algorithm works.

1. Start with a list, LIST, of dyadic cubes. Initially LIST only has one member, namely $[0,1]^n$.

2. Let Q be the last member of LIST. Delete Q from LIST and check property P on Q.

3. If Q passes the test, and LIST is nonempty, go back to Step 2. If LIST is empty, you are done.

4. If Q fails the test, append to LIST all the cubes in the dyadic subdivision of Q. Go back to Step 2.

If this algorithm halts, it means that you have found a partition of $[0,1]^n$ into cubes, all of whom have property P. Hence $[0,1]^n$ has property P as well. If the algorithm never halts, it does not mean that $[0,1]^n$ does not have property P. It just means that you are not going to prove it using your algorithm.

The divide and conquer algorithm described above is closely related to a *depth first search* algorithm. To see the connection, let's think about what happens when $n = 1$, so you just have the binary tree of dyadic intervals. When you perform the subdivision in Step 3 above, you should agree to put the left half of the subdivision before the right half. When you draw the tree in the plane as in Figure 1, the intervals are searched in the following way:

1. Draw a loop around the tree and trace it around clockwise.

2. Whenever a vertex has property P, take a shortcut across the vertex and continue to the next untested vertex along the loop.

Figure 2 shows an example in which the algorithm halts with success.

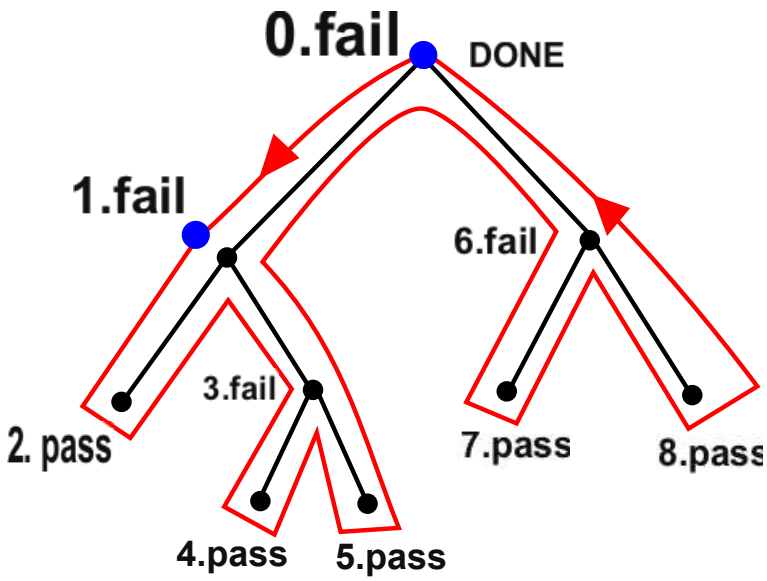

Figure 2: Divide and conquer in action.

If the algorithm doesn't halt, it follows some infinite branch of the tree out towards the end. You could imagine variants where the algorithm halts and declares failure if it reaches a certain depth in the tree.

3 A Connection to Analysis

Here I will give a proof of one of the classic results in real analysis (or point-set topology). Suppose that the unit cube is contained in some union of open balls:

$$[0,1]^n \subset \bigcup_{\alpha} B_\alpha \tag{1}$$

Then $[0,1]^n$ is also contained in some finite sub-collection of the balls:

$$[0,1]^n \subset B_1 \cup ... \cup B_N. \tag{2}$$

This is really the statement that the unit cube is compact.

Here is the proof. Given a cube Q, let $P(Q)$ be the property that Q is covered by a finite number of open balls from our collection. Note that the collection of balls is decided upon in advance. What happens when you run the divide-and-conquer algorithm? If the algorithm halts, then it means $P([0,1]^n)$ is true. In this case, we are done.

If the algorithm does not halt then it burrows down along an infinite branch of the tree. That is, it produces an infinite chain $[0,1]^n \to Q_1 \to Q_2 \to ...$ of dyadic cubes such that $P(Q_j)$ is false for all j. But then consider the single point

$$q = \bigcap_{j=1}^{\infty} Q_j. \tag{3}$$

This point is contained in some open ball from our collection, say $q \in B$. Since B is an open ball, we have $Q_j \subset B$ for all j sufficiently large. (See Figure 3.) But then $P(Q_j)$ is true, and we have a contradiction.

Figure 3:

There is one detail in the proof that requires some background in real analysis. You have to know that the intersection in Equation 3 really is a single point. Doing things coordinatewise, you just have know this for intervals. The claim is then justified by the completeness of the real numbers, or else the least upper bound property.

4 Connection to Algebra

In class I waffled around about how you test that a polynomial is positive on the unit cube, mumbling things about derivative bounds. Let me give a different kind of criterion here. The criterion is something that I came up with myself and have used in a number of situations. I'm sure that it also appears somewhere in the vast literature on polynomials, but maybe it has not been married to a divide and conquer algorithm before. Anyway, the algorithm works like a charm, whoever thought of it first.

First of all, as I said in class, for the case of intervals the method of **Sturm Sequences** works like magic. Look it up. What is nice about the method here is that it works in all dimensions.

Even though Sturm Sequences is the method of choice in one dimension, let me describe the one dimensional case first, just because it is easiest to understand. After I describe the single variable case, I'll explain how it generalizes.

Let

$$F(x) = a_0 + a_1 x + ... + a_n x^n \tag{4}$$

be a polynomial with real coefficients. Here is a criterion for telling that $F > 0$ on $[0,1]$. Define

$$A_k = a_0 + \cdots + a_k. \tag{5}$$

Call F *positive dominant* (or *PD* for short) if $A_k > 0$ for all k.

Lemma 4.1 *Suppose F is positive dominant. Then $F > 0$ on $[0,1]$.*

Proof: The proof goes by induction on the degree of F. The case $\deg(F) = 0$ follows from the fact that $a_0 = A_0 > 0$. Obviously $F(0) > 0$. Let $x \in (0,1]$. We have

$$F(x) = a_0 + a_1 x + x_2 x^2 + \cdots + a_n x^n \geq$$

$$a_0 x + a_1 x + a_2 x^2 + \cdots + a_n x^n =$$

$$x(A_1 + a_2 x + a_3 x^2 + \cdots a_n x^{n-1}) = xG(x) > 0$$

Here $G(x)$ is positive dominant and has degree $n-1$. ♠

Given a polynomial F, we define

$$F_0(x) = F(x/2), \qquad F_1(x) = F(1 - x/2). \tag{6}$$

Lemma 4.2 *$F \geq 0$ on $[0,1]$ iff $F_0 \geq 0$ on $[0,1]$ and $F_1 \geq 0$ on $[0,1]$.*

Proof: Let $A_0(x) = x/2$ and $A_1(x) = 1 - x/2$. Then A_0 is a bijection from $[0,1]$ to $[0,1/2]$ and A_1 is a bijection from $[0,1]$ to $[1/2,1]$. So, $F \geq 0$ on $[0,1/2]$ iff $F_0 \geq 0$ on $[0,1]$ and $F \geq 0$ on $[1/2,1]$ iff $F_1 \geq 0$ on $[0,1]$. ♠

The polynomial pair $\{F_0, F_1\}$ is defined to be the *subdivision* of F. Instead of subdividing the intervals, we "subdivide" the polynomials. This amounts to the same thing. We then perform the divide and conquer algorithm as above. The algorithm halts if and only if $F > 0$ on $[0,1]$. The reason why it works is that the subdivision process tends to increase the chances that a polynomial is positive dominant, because it tends to put more weight on the earlier terms in the polynomial.

Now consider the multivariable case, where the variables are $x_1, ..., x_n$. I'll use multi-index notation:

$$x^I = x_1^{i_1} ... x_n^{i_n}, \qquad I = (i_1, ..., i_n). \tag{7}$$

Any polynomial $F \in \boldsymbol{R}[x_1, ..., x_n]$ can be written succinctly as

$$F = \sum A_I X^I, \qquad A_I \in \boldsymbol{R}. \tag{8}$$

If $I' = (i_1', ..., i_n')$ we write $I' \leq I$ if $i_j' \leq i_j$ for all $j = 1, ..., n$. We call F *positive dominant* if

$$\sum_{I' \leq I} A_{I'} > 0 \qquad \forall I, \tag{9}$$

Lemma 4.3 *Suppose that F is positive dominant. Then $F > 0$ on $[0,1]^n$.*

Proof: The 1 variable case is Lemma 4.1. In general, write

$$F = f_0 + f_1 x_n + ... + f_m x_n^m, \qquad f_j \in \boldsymbol{R}[x_1, ..., x_{n-1}]. \tag{10}$$

Let $F_j = f_0 + ... + f_j$. Since F is positive dominant, we get that F_j is positive dominant for all j. By induction on n, we get $F_j > 0$ on $[0,1]^{n-1}$. But now, if we hold $x_1, ..., x_{n-1}$ fixed and let $t = x_n$ vary, the polynomial $g(t) = F(x_1, ..., x_{n-1}, t)$ is positive dominant.. Hence, by Lemma 4.1, we get $g > 0$ on $[0,1]$. Hence $F > 0$ on $[0,1]^n$. ♠

Now we have to define the analogue of subdivision. Basically, we replace $F(x_1, ..., x_n)$ by the new polynomial

$$F(x_1', ..., x_n'), \tag{11}$$

where either $x_j' = x_j/2$ or $x_j' = 1 - x_j/2$. This produces 2^n new polynomials. The original polynomial F is positive on $[0,1]^n$ if and only if all the new polynomials are positive on $[0,1]^n$. This list of 2^n polynomials is declared to be the subdivision of F. The n dimensional version then works just like the 1 dimensional version. Again, the algorithm halts if and only if $F > 0$ on $[0,1]^n$.

5 Connection to Geometry

Now I'll explain something about how one can use a divide and conquer algorithm to find periodic billiard paths in triangles. A periodic billiard path in a triangle determines an infinite repeating sequence using the symbols $\{1,2,3\}$. For instance, the periodic billiard path shown on the left half of Figure 4 creates the sequence 123123... The periodic billiard path on the right creates the sequence 132123....

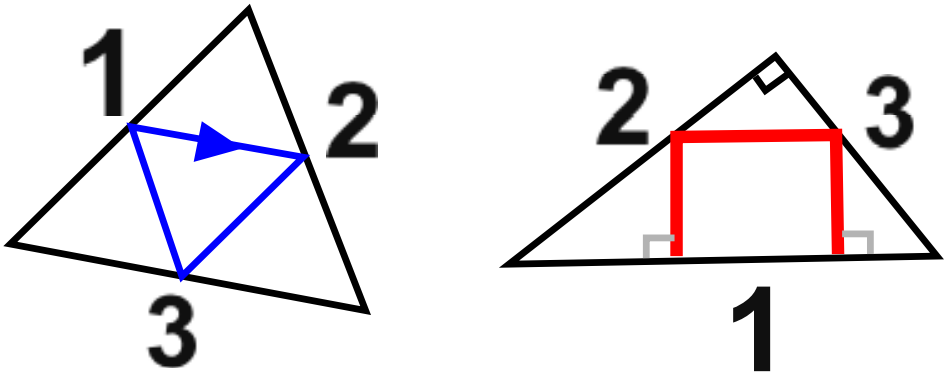

Figure 4:

A periodic billiard path in a triangle is called *stable* if it also exists in all nearby triangles. (I didn't explain this in class, but below I will connect it up to what I did say in class.) The periodic billiard path on the left is stable and the periodic billiard path on the right is not stable: It only works in right triangles. The algorithm I will describe searches for stable periodic

billiard paths, though at some point we (Pat Hooper and I) made a variant for unstable periodic billiard paths.

Let me first describe a combinatorial criterion for stability. Label the edges of the infinite hexagonal grid by 1, 2, and 3 depending on the slope of the edge. Figure 5 shows what I mean.

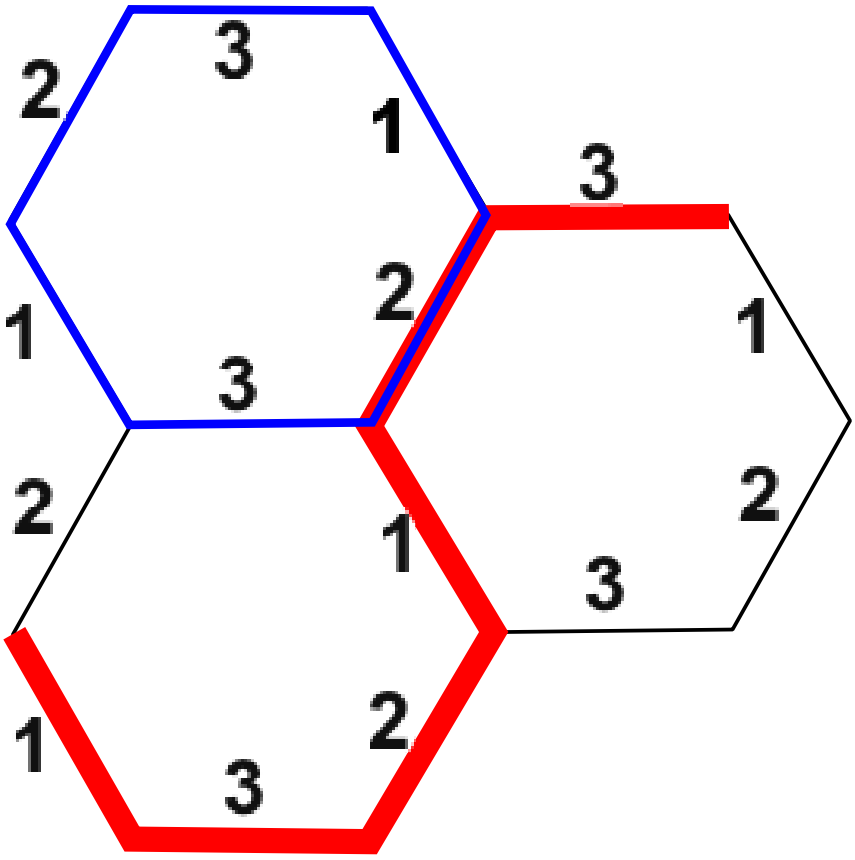

Figure 5: 123123 in blue and 132132 in red.

If you have a periodic string, like 123123 you can draw the corresponding path on the hexagonal grid, using the labelings of the sides. If the string corresponds to a periodic billiard path, then the billiard path is stable if and only if the string make a closed loop! Notice that the blue loop, corresponding to the left side of Figure 4 is closed and the red loop, corresponding to the right hand side, is not.

Call the string S *combinatorially stable* if the corresponding path on the hexagonal grid is a closed loop. What I have said so far is: If S corresponds

to a periodic billiard path on the triangle T and S is combinatorially stable, then the billiard path is stable.

Suppose that S is combinatorially stable. Here is how to test that S corresponds to a periodic billiard path on a triangle T. You reflect T in its sides, repeatedly, for all but the last digit of the string. This produces a kind of "carpet" of triangles. The combinatorial stability condition guarantees that the first and last sides are parallel. (This connects with what I said in class.) Call the carpet *straight* if there is a straight line which stays in the carpet and connects equivalent points on the first and last sides. Here *equivalent* means that the points are paired up by the translation which maps the first side to the last side. Figure 6 shows all this for some acute triangle T and the string 123123.

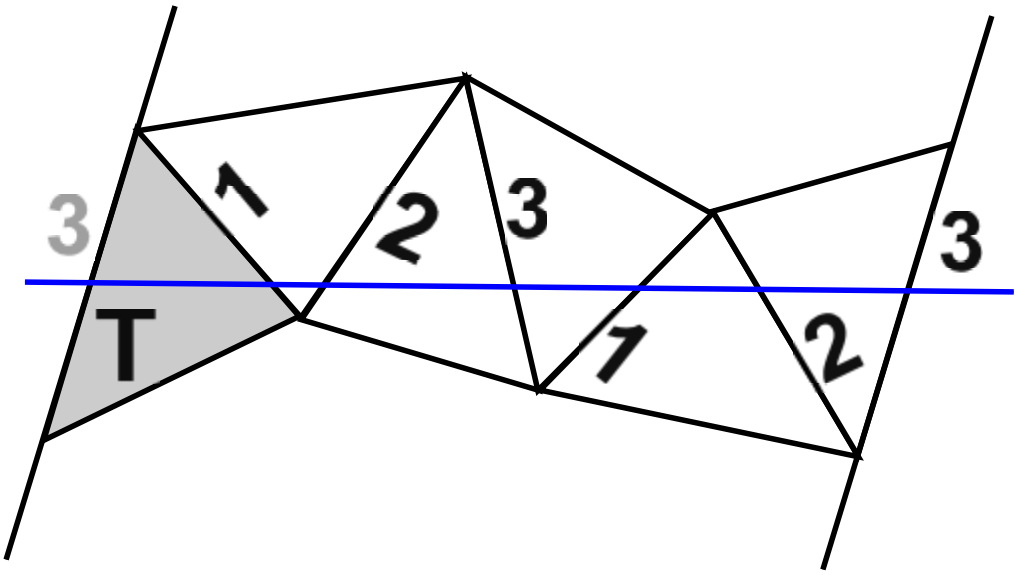

Figure 6: A straight carpet for 123123.

If the string S is combinatorially stable and if the corresponding carpet T_S is straight, then T has a stable periodic billiard path described by S. In this case, call S a *good string* for T.

For an arbitrary string S', not necessarily combinatorially stable, call the carpet $T_{S'}$ *crooked* if there is no straight line segment, contained in the carpet, which connects the first and last sides. If $T_{S'}$ is crooked then S' cannot be the substring of a good string for T. Further reflections are not going to straighten the thing out.

Finally, we get to the depth first search algorithm. We take the binary tree of possible strings, as shown in Figure 7.

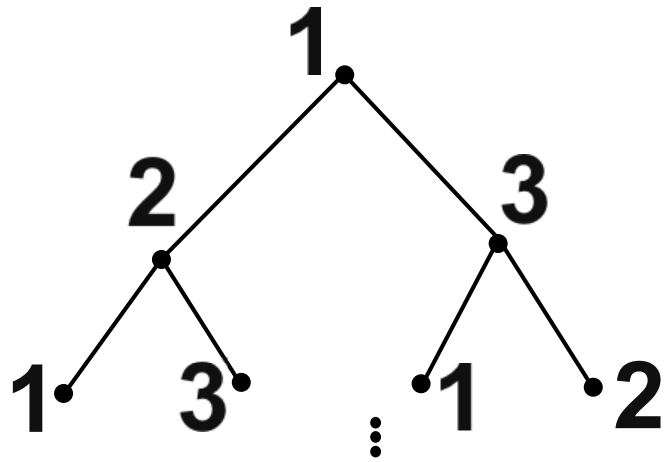

Figure 7: The tree of potential strings

We perform the following algorithm, which looks for a periodic billiard path corresponding to a string of length at most L. The algorithm runs with some fixed triangle T specified in advance.

1. Start with a list LIST of vertices. Initially LIST has one member, the top vertex.
2. Let S be the last member of LIST. Delete S from LIST.
3. If S has length L and LIST is not empty go to Step 2.
4. If T_S is crooked and LIST is not empty, go to Step 2.
5. If S has length less than L and T_S is not crooked and S is not combinatorally stable then append to LIST the two vertices beneath S and go to Step 2.
6. If S is combinatorially stable and T_S is straight, HALT with success.
7. If LIST is empty HALT with failure.

You could imagine a variant in which you don't halt at Step 6 but rather you just append to some new list of winners the string that you have found. With this variation, the algorithm will get to end end of the tree and find all the stable periodic billiard paths of length at most L.