Theoretical Computers and Diophantine Equations *

Alvin Chan
Raffles Junior College

&

K. J. Mourad
Nat. Univ. of Singapore

§1. Introduction

This paper will discuss algorithms and effective computability, and how these concepts relate to Turing Machines and Diophantine sets and relations. The link between the two will also be examined followed by an account of several important diophantine relations.

§2. Recursiveness and computability

Recursive functions can be informally thought of as functions which are effectively computable. Simply put an effectively computable function, f, is one for which there exists an effective procedure (an algorithm) which calculates the value of $f(x)$. An effectively computable function must satisfy the following conditions.

(i) If the procedure is supplied with a value x that is in the domain of f, then after a finite number of steps the procedure must stop and produce a value for $f(x)$.

(ii) If the procedure is supplied with a value that is not in the domain of f then the procedure might carry on never halting, or it might get stuck at some point. Most importantly it must *never pretend* to produce a value $f(x)$.

This procedure could then be visualised in modern day terms as a digital computer executing a program to determine the value of an element in the range of a function. Included too in the definition of an effective procedure are several restrictions that must *never* be imposed.

(i) There must be no limit imposed on the size of the input although it must be a natural number.

---

* Paper presented to the Science Research Congress 1990 as part of the 1990 Science Research Programme organised jointly by National University of Singapore and Ministry of Education; supervisor K. J. Mourad.

(ii) Although after a certain finite number of steps the procedure must stop and produce a value for $f(x)$, there is no restriction imposed on the number of such steps.

(iii) There must be no restriction imposed on the amount of memory space the procedure utilises in calculating the value of $f(x)$.

These restrictions become significant when we compare the concept of effective computability with that of practical computability. For example a function might be deemed practically computable if it could be evaluated by a digital computer in 5 hours for an effectively computable function, no bound is imposed on the time taken to calculate it as long as it is finite.

§3. Turing machines

A way to defining recursiveness was formulated by Alan Turing in 1936 in terms of imaginary computing machines known as Turing machines.

Turing machines are basically computing machines with rudimentary structures. We can start by imagining such machines to consist of a box with a long length of tape going through it. This tape is divided into squares, and on each square is either a 0 or 1. This tape is infinite in both directions, however only a finite number of squares may contain ones. This tape initially contains the input of the machine and ultimately the output of the machine. At intermediate stages it serves as working space for the calculation.

The workings of a Turing machine may be defined as follows. It is only capable of examining one square at a time, and the box of the Turing machine consists of a finite list of instructions $(q_1, q_2 ... q_n)$. Each single instruction indicates 2 possible courses of action, one to be followed if the square being scanned contains a 1, the other if the square contains a 0. In either case a course of action consists of three steps. Firstly, the square is scanned and then a new symbol (either a 1 or 0) is written on the same square. Secondly, the machine moves the tape either a square to the left or to the right. Thirdly, a new instruction is specified. According to this description a Turing machine might be defined as a function M.

Definition:

A Turing machine is a function M such that for some natural number n,

$$\text{domain of} \quad M \subseteq \{0,1...n\} \times \{0,1\}$$
$$\text{range of} \quad M \subseteq \{0,1\} \times \{L,R\} \times \{0,1...n\}$$

Let us take for example $M(5,0) = (1,R,7)$. This means that Turing machine M upon reading a square which contains a 0 while executing instruction 5 will write a new symbol 1 on that square and then move the tape one square to the right. Then the machine will move on to execute instruction 7.

Finally we are able to define recursiveness in terms of such a machine. Let x be a string of 1s. Then $[x_1 0 x_2 0...0 x_k]$ would be k strings of 1s separated by 0s. Let this be the input/output format of machine M. A function is then said to be recursive if there exists a machine M such that whenever we start M at instruction q_0 scanning the leftmost symbol of $[x_1 0 x_2 0...0_k]$ then if $f(x_1...x_k)$ is defined, M will eventually halt scanning the leftmost symbol of the output format $[f(x_1...x_k)]$. If $f(x_1...x_k)$ is undefined then M will never halt.

§4. Alternative ways of looking at Turing machines

Besides being thought of as a function, a Turing machine could also be thought as a table of quintuples (5-tuples) in the form $(i,\ \alpha,\ \beta,\ x,\ j)$ with α being the current symbol and β the symbol (possibly equal to α) to replace α, x being the direction the tape is going to take and i and j representing the current and future instruction. We can then define a k-ary relation, P,as follows: Turing machine T eventually halts if and only if $< x_1...x_k > \in P$. Next we can code configurations of T by $< l,n,r >$ where l is the natural number whose binary representation is actually the string of numbers to the left of the square being scanned, n is the number of the instruction being executed and r is the natural number which encodes the string of numbers to the right of and including the number being scanned (read from right to left).

Next let the set S_Q be the set of all $< l,n,r,l',n',r' >$ such that a quintuple Q from the table of machine T acts on $< l,n,r >$ to produce $< l',n',r' >$ (the next configuration). We see then that if Q is in the form: $(i,\ \alpha,\ \beta,\ x,\ j)$

$$\text{then if } x = R \text{ then } < l,n,r,l',n',r' > \in S_Q \iff n = i \ \wedge n' = j$$
$$\wedge\ l' = 2l + \beta \ \wedge r = 2r' + \alpha$$

and if $x = L$ then $< l, n, r, l', n', r' > \in S_Q \iff n = i \wedge n' = j \wedge$
$[(l = 2l' \wedge r' = 2(r + \beta - \alpha)) \vee (l = 2l' + 1 \wedge r' = 2(r + \beta - \alpha) + 1)]$

Hence, we can see how the configurations of a Turing machine computation can be represented by a series of equations. We will discuss this further in what follows. First we deal with a fundamental question concerning algorithms.

§5. The halting problem

As discussed earlier on, a Turing machine if put to the task of calculating a value of $f(x)$ will halt and produce a value for $f(x)$ if it is defined, but will carry on attempting to find $f(x)$ never halting if $f(x)$ is undefined. The problem then is being sure when $f(x)$ is undefined, to be absolutely sure that $f(x)$ is undefined would mean that the Turing machine calculating a value for $f(x)$ would never halt. But the catch is that we can ever be absolutely certain that the machine never halts, unless we wait for an infinite length of time! Imagine a digital computer that we set to finding solutions to a certain problem. After a time of 10 years no solution has been found but this does mean that the machine will not find a solution within the next second in time. On the other hand we might for another thousand years and still no solution would be found.

The question then arises: Is there an algorithm by which we could determine whether a Turing machine halts? First let us consider a simple program in BASIC (digital computer programs can represent Turing machines and vice versa.)

```
10  input x
20  if x < 0 then goto 10
30  let y = y + 1
40  if x + y = 5 then goto 60
50  goto 30
60  end
```

For such a simple program, we see that an algorithm can be found to determine if it halts or not. Simply check the value of x. If it exceeds 4

then the program will keep searching unsuccessfully for a solution to $x+y$ and will never halt. If the value of x is 4 or smaller then the program will find a value for y and halt. Although for such a simple program we are able to determine whether it halts this is generally not possible for more complicated programs and for Turing machines.

Theorem:

There is no algorithm for testing whether a Turing machine M will eventually halt when put to the task of evaluating $f(x)$ for given x.

Proof:

Let us begin by taking $T_1, T_2 ... T_k ...$ to be a list of all possible Turing machines which calculate all possible recursive functions. Such a listing is possible since each Turing machine can be represented by a finite set which in turn can be represented by the binary representation of an integer k. Let $x_1, x_2 ... x_n ...$ be a list of various inputs that we feed into the various machines. (Note that this list of inputs contains potentially infinitely many terms). Assuming that we have an algorithm for determining whether a machine halts, we then use a 1 to represent that the machine halts on a given input and a zero to represent that it does not. We can then construct a table to show which machines halt or do not halt on various inputs.

	x_1	x_2	x_3	x_4	x_5	...	x_n	...
T_1	1	0	1	1	0			
T_2	0	0	1	0	0			
T_3	1	1	0	1	1			
T_4	1	0	0	0	1			
T_5	0	1	1	1	0			
⋮								
T_k								
⋮								

Next we take the diagonal of the table as shown and invert it (0s become 1s and vice versa). This means that the diagonal (1 0 0 0 0) would become (0 1 1 1 1). From this we notice a very interesting fact. This inverted diagonal represents a new Turing machine not found in the list $T_1 ... T_k ...$! This is because the p'th term (where p is a positive integer) in the string of ones and zeros following machine T_p will always be different

from the pth term in the string of ones and zeros of the new machine. For example the third term of machine T_3 is a zero while the third term of the new machine is a 1.

We see then where the contradiction lies. Assuming that we are able to the determine halting and that $T_1 ... T_n ...$ is a list of *all possible* Turing machines, how is it that we able to obtain a yet another Turing machine not in the list. Thus we must conclude that we are unable to determine halting. (Note that the set K where K is the domain of recursive function f is *recursively enumerable* - this means that we can be sure a value x is in K only if the Turing machine halts, if it has not halted then we do not know if that value of x is in set K.)

§6. Diophantine equations and sets

Diophantine equations are simply equations consisting of polynomials whose variables range over the set of integers. From this simple description we can then define Diophantine sets.

Definition:

A set S of ordered n-tuples is called Diophantine if there is a polynomial $P(x_1 ... x_n, y_1 ... y_m)$ with integer coefficients and where $m \geq 0$ such that an n-tuple $< x_1 ... x_n >$ then belongs to the Diophantine set S iff there exists integers $< y_1 ... y_m >$ such that $P(x_1 ... x_n, y_1 ... y_m) = 0$. In other words, $S = \{< x_1 ... x_n > \ (\exists y_1 ... y_m)[P(x_1 ... x_n, y_1 ... y_m) = 0]\}$. Alternatively the n-tuple could be thought of as an n-ary relation where S is the set of objects in this Diophantine relation.

From this examination of Diophantine sets we make several observations.

(i) There is no algorithm for testing if there exists an m-tuple $< y_1 ... y_m >$ such that for a given n-tuple $< x_1 ... x_n > P(x_1 ... x_n, y_1 ... y_m) = 0$.

(ii) Any diophantine set S is recursively enumerable.

Let us examine whether we could possibly find an algorithm to determine the existence of solutions to diophantine equations.

We can take the Diophantine set S and define its characteristic function, $G_s : Z^n \longrightarrow Z$, where Z denotes the set of integers, such that $G_s(x_1 ..., x_n) = 0$ if the n-tuple is in the set S and $G_s(x_1 ..., x_n)$ to be undefined otherwise. Next we take Turing machine A to be the machine which evaluates the recursive function G_s. An algorithm for evaluating such a function consist of arranging in sequence all possible combinations

of the m-tuple $< y_1 ..., y_m >$ and evaluating the polynomial P until (if ever) the value 0 is obtained, after which we conclude that this n-tuple $< x_1 ... x_n >$ is in S. If we could determine if there existed an m-tuple such that $P(x_1 ..., x_n, y_1 ..., y_m) = 0$ for a given n-tuple, it would mean finding an algorithm for determining if $G_s(x_1 ..., x_n) = 0$ or is undefined, which would mean determining whether Turing machine A halts! We have shown that there is no effective method to do the latter so if every Turing machine could be so represented we could then conclude that there is no method for determining the existence of solutions of diophantine equations.

First, recall the previous section in which the various Turing machine configurations were shown to be related by polynomial equations. From this we can in fact show that every Turing machine is represented by a diophantine equation and vice versa. The proof due to the combined efforts over many years of Y. Matijacevič, Julia Robinson, Martin Davis, and Hilary Putnam will not be shown here but instead the reader is referred to [DAVIS 1977].

Hence the unsolvability of the halting problem now directly implies the lack of an algorithm for determining the existence of solutions. If we wanted to make this fact more explicit, we could again use the diagonal argument, taking the $T_1 ... T_n ...$ to represent all possible diophantine equations and $x_1 ... x_n ...$ to represent all the various parameters for which we wanted to test for the existence of solutions. Again a 1 would represent existence while a zero would represent non-existence. We easily see how the second observation comes in. As there is no algorithm for determining the existence of an m-tuple for a given n-tuple, the only way of determining if an n-tuple is in set S would be to polynomial P until a value of zero is obtained. But while trying out different combinations of m-tuples in an attempt to evaluate P to zero, we would never know if that particular n-tuple is in set S. Thus we can only be sure that an n-tuple is in set S, but we can never be sure that an n-tuple is not in set S.

Although there is no algorithm to test whether there exists solutions to a polynomial for a given set of parameters, for simple polynomials such as linear equations there exists such algorithms. For linear equations we see that there is a simple algorithm for testing the equation $ax + by = c$ for the existence of solutions in x and y. Simply take the triple $< a, b, c >$ that we put into the equation and evaluate the greatest common divisor (g.c.d.) of a and b if c is divisible by this g.c.d., then there exists solutions in x and y. We also note other interesting facts about linear equations. For a triple $< a, b, c >$ for which there exists solutions, we can simply

divide a, b and c by the g.c.d. to obtain a new triple. The new triple will have the same solution set in x and y as the original triple $< a, b, c >$.

From our excursion through the concept of Diophantine sets, we notice certain striking similarities to Turing machines.

(i) Both have simple exceptions to the unsolvability of the halting problem and the non-existence of an algorithm which tests a polynomial for the existence of solutions with given parameters.

(ii) We have seen how the non-existence of an algorithm for testing a given polynomial for the existence of solutions with given parameters is related to the halting problem.

(iii) Both Diophantine sets and the domains of the recursive functions which Turing machines evaluate are recursively enumerable.

§7. Some Diophantine relations

For a period of time, mathematicians had been trying to prove that the relation $z = x^y$ was diophantine. The difficulty of such a proof was that combinations of equations which were already known to be Diophantine had to be used to encode the exponentiation relation, such that one of the solutions sets of this combination of diophantine relations would grow exponentially in relation to another solution set. In order to better appreciate the difficulty let us consider the example of $2x + 3y = 0$. It is obvious that the solution set to such an equation would be $3a$ (where a is an integer) for x and $2a$ for y. Clearly linear equations cannot encode exponentiation.

The proof for exponentiation was finally conquered in the nineteen seventies. No proof is given here, rather the reader should refer to any of the following for a detailed proof: MATIJACEVIČ [1972], DAVIS [1971, 1973]. What the proof involved was the using of Pell's equations to encode exponentiation.

The proving of exponentiation as being Diophantine was extremely significant as it confirmed that the bounded quantifier theorem (refer to DAVIS [1973]) was indeed correct. The proof of this theorem had been done earlier assuming that the exponentiation was diophantine. This theorem expanded the language of Diophantine predicates and enabled Diophantine sets to be defined by bounded existential quantifiers $(\exists y)_{\leq x}$ which means $(\exists y)\ (y \leq x)$ and bounded universal quantifiers $(\forall y)_{\leq x}$ which means $(\forall y)\ (y \leq x)$. Putting all these results together gave the full solu-

tion to Hilbert's tenth problem attributed to the four people mentioned above.

With these expanded means of defining Diophantine sets let us then take a look at the set P of prime numbers. The set can be defined as

$$x \in P \longleftrightarrow x > 1 \wedge (\forall y, z)_{\leq x}[y \cdot z < x \vee y \cdot z > x \vee y = 1 \vee z = 1]$$

The proof that exponentiation was Diophantine also enabled the relations $m = \binom{n}{k}$ and $m = n!$ to be confirmed as being Diophantine. A proof, assuming that exponentiation is diophantine, will now be given.

Proof:

First let us write $\phi(k, u, w)$ for the coefficient of u^k in the u-ary (base u) expansion of w. We see then that the relation $q = \phi(k, u, w)$ is Diophantine as it can be expressed in simultaneous solvability in $\mathcal{N}$, the set of integers greater than or equal to 0, of these four conditions:

(i) $q + a + 1 = u$ (i.e. $q < u$)
(ii) $p + b + 1 = u^k$ (i.e. $p < u^k$)
(iii) $c \cdot u^{k+1} = r$ (i.e. $r \equiv 0 \pmod{u^{k+1}}$)
(iv) $w = p + q^k + r$

Then if $u > 2^n$ the binomial $(u+1)^n = \sum_{i=0}^{n} \binom{n}{i} u^i$. This shows that $\binom{n}{k}$ is simply the coefficient of u^k in the u-ary expansion of $(u+1)^n$. Hence,

$$m = \binom{n}{k} \longleftrightarrow m = \phi(k, 2^n + 1, (u+1)^n).$$

We note that the conjunction of two polynomial equations $P = 0$ and $Q = 0$ is equivalent to the one equation, $P^2 + Q^2 = 0$. Also solvability in $\mathcal{N}$ can be expressed by searching for solutions which are the sum of four squares of integers (using the famous theorem of Lagrange which states that being an element of $\mathcal{N}$ is equivalent to this condition). Thus $m = \binom{n}{k}$ is Diophantine.

Next, for the relation $m = n!$. It will be shown below that when $r > \frac{(n!+1)(n)(n-1)}{2}$ then $n! \leq \frac{r^n}{\binom{r}{n}} \leq n! + 1$ holds true. Thus

$$m = n! \longleftrightarrow \exists r \exists s \exists t \exists u \exists v \{r = n^n \ \& \ u = \binom{r}{n}$$
$$t = r^n \ \& \ u \cdot m + s = t \ \& \ t + v = u(m+1)\}.$$

This shows that $m = n!$ is Diophantine.

Proposition:

When $r > \frac{(n!+1)(n)(n-1)}{2}$ then $n! \leq \frac{r^n}{\binom{r}{n}} \leq n! + 1$.

Proof:

We examine the inequality $\frac{r^n}{\binom{r}{n}} \leq n! + 1$ (note that $n! \leq \frac{r^n}{\binom{r}{n}}$ always holds).

Manipulating the inequality:

$$\frac{r^n}{\binom{r}{n}} \leq n! + 1 \leftrightarrow \frac{r^n n!(r-n)!}{r!} \leq n! + 1$$

$$\leftrightarrow \frac{r^{n-1}}{(r-1)(r-2)\ldots(r-(n-1))} \leq \frac{n!+1}{n!}$$

$$\leftrightarrow \sum_{m=0}^{n-1} \frac{\alpha_n(m)}{r^m} \geq \frac{n!}{n!+1}$$

(where $\alpha_n(m)$ is the coefficient of r^{-m} in $\frac{r!}{r^{n-1}n!}$)

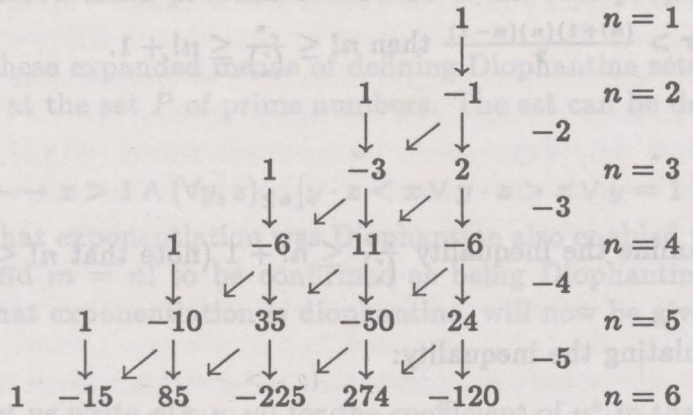
$$\leftrightarrow 1 + \sum_{m=1}^{n-1} \frac{\alpha_n(m)}{r^m} \geq \frac{n!}{n!+1}$$

$$\leftrightarrow \sum_{m=1}^{n-1} \frac{\alpha_n(m)}{r^m} \geq -\frac{1}{n!+1}$$

$$\leftrightarrow \frac{\alpha_n(1)}{r} + \sum_{m=2}^{n-1} \frac{\alpha_n(m)}{r^m} \geq -\frac{1}{n!+1}$$

From this, we approximate for r by using $\frac{\alpha_n(1)}{r} \geq -\frac{1}{n!+1}$. This is possible as $\sum_{m=2}^{n-1} \frac{\alpha_n(m)}{r^m}$ is small as compared to $\frac{\alpha_n(1)}{r}$ because of the much larger denominator in the former term.

Next we evaluate the values of $\alpha_n(m)$ for $n = 2$ to 6 and $m = 0$ to $n-1$. We find that they fall in the pattern below:



The vertically downward arrows indicate that the number above the arrow is multiplied by the number beside the arrow and then added to the number above the diagonal arrow to obtain the result below both arrows, eg. in the row $n = 4$ $-6 = (1)(-3) + (-3)$.

From this pattern we discern certain interesting observations

(i) The coefficients $\alpha_n(1)$ are always negative and are evaluated by the formula $\frac{-n(n-1)}{2}$.

(ii) To approximate the bound of r from $\frac{\alpha_n(1)}{r} > -\frac{1}{(n!+1)}$ it is enough to show that, for r large enough, $\sum_{m=2}^{n-1} \frac{\alpha_n(m)}{r^m} > 0$ (using: $a > c$ and $b > 0$ implies $a + b > c$).

Proof of observation 1:

Let $n = k$ then assume $\alpha_k(1) = -\frac{k(k-1)}{2}$ let $n = k+1$ then $a_{k+1}(1) = -k + -\frac{k(k-1)}{2} = -\frac{((k+1)-1)(k+1)}{2}$ Hence, by induction $\alpha_n(1)$ is indeed $\frac{-n(n-1)}{2}$.

Now we show that the requirement in the second observation can be met when r is large enough ($r > \frac{(n)(n-1)}{2}$ will suffice). To prove that $\sum_{m=2}^{n-1} \frac{\alpha_n(m)}{r^m} > 0$ it suffices to show that $\frac{\alpha_n(x)}{r^x} > -\frac{\alpha_n(x+1)}{r^{x+1}}$ when $x = 2y$ (since $\frac{\alpha_n(x)}{r^x}$ is always positive when x is odd).

(1) $\alpha_n(x) = -(n-1)\alpha_{n-1}(x-1) + \alpha_{n-1}(x)$ hence $\alpha_n(x) > \alpha_{n-1}(x)$.

Therefore

$$(\frac{(n)(n-1)}{2})\alpha_n(x) > (\frac{(n)(n-1)}{2})\alpha_{n-1}(x).$$

(2) $-\alpha_n(x+1) = \sum_{k=1}^{n-1} \alpha_{n-k}(x)(n-k)$. This follows inductively using the recurrence relation.

$$\sum_{k=1}^{n-1} \alpha_{n-1}(x)(n-k) > \sum_{k=1}^{n-1} \alpha_{n-k}(x)(n-k) \quad \text{and hence}$$

$$(\frac{n(n-1)}{2})\alpha_{n-1}(x) > -\alpha_n(x+1)$$

From 1) and 2) we have $(\frac{(n)(n-1)}{2})\alpha_n(x) > -\alpha_n(x+1)$. If $r > \frac{(n)(n-1)}{2}$ then $\frac{r\alpha_n(x)}{r^{x+1}} > -\frac{\alpha_n(x+1)}{r^{x+1}}$. Hence $\frac{\alpha_n(x)}{r^x} > -\frac{\alpha_n(x+1)}{r^{x+1}}$ when $r > \frac{(n)(n-1)}{2}$. It now follows that $\frac{(n!+1)(n)(n-1)}{2}$ gives us our required bound.

§5. Conclusion

We have seen how polynomials can represent Turing machines. Thus polynomials which look simple at first are actually more complex than we had thought, because of the large amount of data that they can encode. A most intriguing question for further thought would be to determine under what conditions the simple polynomials for which we can find an algorithm to determine the existence of solutions become truly complicated (that is when we can no longer determine if there exist solutions to the polynomial).

References

[1] M. DAVIS, An explicit Diophantine definition of the exponential function, *Comm. Pure Appl. Math.*, **24** (1971), 137-145.

[2] M. DAVIS, Hilbert's tenth problem is unsolvable, *Am. Math. Monthly*, **80** (1973), 233-269.

[3] M. DAVIS, Unsolvable problems, *Handbook of Mathematical Logic*, (1977), 567-594, edited by J. Barwise (North Holland, Amsterdam).

[4] Y. MATIJACEVIČ, Diophantine sets, *Uspehi Mat. Nauk*, **27** (1972), 185-222. [English translation in Russian Math. Surveys, 27 124-164].