15-354: CDM	**K. Sutner**
Assignment 3	Due: **Sep. 22, 2023, 24:00**.

1. Semidecidable Sets and Computable Functions (40)

Background

We defined semidecidable sets as a generalization of decidable sets: on a Yes-instance the "semidecision algorithm" terminates, but on a No-instance it keeps running forever. There are many alternative characterizations that describe more directly the relationship between semidecidable sets and partial computable functions.

By an enumeration of $A \subseteq \mathbb{N}$ we mean a partial function $f : \mathbb{N} \nrightarrow \mathbb{N}$ so that the range of f is A. For simplicity, we will assume that the support of f is either all of $\mathbb{N}$ or some initial segment $\{0, 1, \ldots, n-1\}$. So

$$A = \{ f(i) \mid i < N \} = f(0), f(1), f(2), \ldots$$

where $N = n$ or $N = \omega$. Note that we allow $n = 0$ corresponding to $A = \emptyset$. An enumeration is repetition-free if f is injective. A set is recursively enumerable (r.e.) if it can be enumerated by a computable function f.

Task

Assume that $A \subseteq \mathbb{N}$. Show the following.

A. All finite sets are recursively enumerable.

B. The set of primes is recursively enumerable.

C. The set of prime twins is recursively enumerable.

D. A is semidecidable iff it is recursively enumerable.

E. A is semidecidable iff it is recursively enumerable with a repetition-free enumeration.

F. Suppose A is infinite. Then A is decidable iff it is recursively enumerable with a strictly increasing enumeration.

Comment

Don't try to argue formally in terms of register machines, just use computability in the intuitive sense, much the way you would describe a solution to a problem in an algorithms class.

Note that it is currently unknown whether there are infinitely many prime twins—but that does not affect part (C).

Solution: Semidecidable Sets and Computable Functions

Part A: Finite

We can simply hardwire the finite set A into the algorithm that "computes" the enumeration (computes here just means: performs a table-lookup). More precisely, there is a, say, strictly increasing list $a_0, a_1, \ldots, a_{n-1}$ of the elements of A, where n is the cardinality of A. The enumeration maps $i \mapsto a_i$ for $i < n$, and is undefined otherwise.

Part B: Primes

It is not hard to see that primality is decidable (in fact, primitive recursive and, as we now know, polynomial time) and that the function `nextprime` is computable. But then we can compute the nth prime as follows:

```
p = 2;
for( i = 0; i < n; i++ )
     p = nextprime(p);
return p;
```

We assume 0-indexing here.

Part C: Prime Twins

Let's call the last program `nthprime`: on input n return the nth prime. Also assume we have a program `prime` that tests primality. The following (atrocious) program returns the nth prime twin (first component only, if you want both use a pairing function).

```
c = 0;
k = 1;
while( c < n )
     while( !prime(nthprime(k)+2) ) k++;
     c++;

return nthprime(k);
```

Sadly, at the time of this writing, no one knows whether this program halts for all n.

Part D: Enumeration

We may safely assume that A is infinite.

If A is r.e., to semidecide membership of $x \in A$, we can simply "run" the enumeration: if x appears, halt, otherwise keep running forever. Since f is computable, this is a semidecision procedure.

For the opposite direction, suppose $\mathcal{A}$ is a semidecision algorithm for A. We orgnize the generating algorithm in stages s (there is an outer loop that executes all stages one after the other). At stage s, we run $\mathcal{A}$ on all $x < s$ for at most s steps. If $\mathcal{A}$ converges on z, we add z to the list of already enumerated elements.

As written, this method repeats each element of A infinitely often, but that is allowed according to our definition.

Part E: Repetition-Free Enumeration

We can use exactly the same argument as in the last part, except that we keep track of a list all already discovered elements of A. Whenever a potentially new element z pops up, we first check against the list.

Part F: Monotonic Enumeration

Now suppose A is decidable. Again think of the enumeration as a list, initially empty, and proceed in stages. At stage s we run the decision algorithm for A on s. If the algorithm returns Yes we append s to the list, otherwise we do nothing (recall that the decision algorithm for A must halt on any input).

For the opposite direction suppose a_s is a monotonic enumeration of A. Given x, to decide membership in A, find the unique s such that either $x = a_s$ or $a_s < x < a_{s+1}$: this can be done by a brute-force search (which must terminate!). Return Yes or No accordingly.

2. The DASZ Operator (30)

Background

For this problem, consider non-decreasing lists of positive integers $A = (a_1, a_2, \ldots, a_w)$. We transform any such list into a new one according to the following simple recipe:

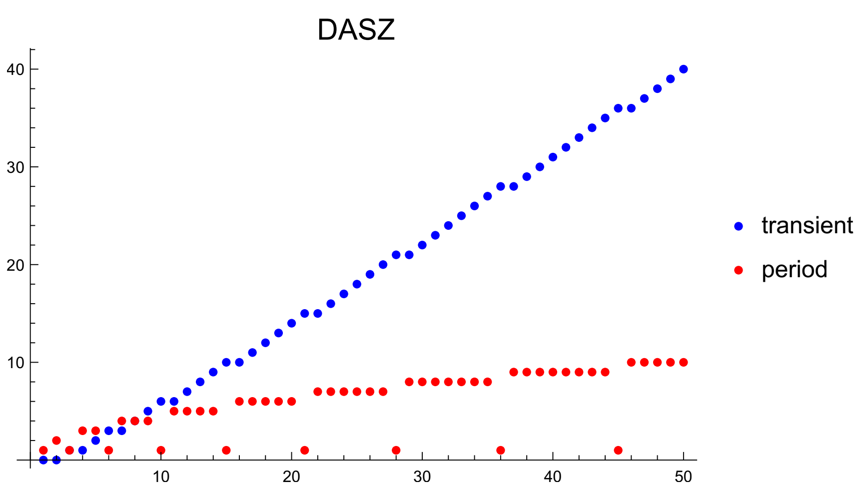
- Subtract 1 from all elements.
- Append the length of the list as a new element.
- Sort the list.
- Remove all 0 entries.

We will call this the DASZ operation (decrement, append, sort, kill zero) and write $D(A)$ for the new list (note that D really is a function). For example, $D(1,3,5) = (2,3,4)$, $D(4) = (1,3)$ and $D(1,1,1,1) = (4)$.

A single application of D is not too fascinating, but things become interesting when we iterate the operation: as it turns out, $D^t(A)$ always has a finite transient (and period), no matter how A is chosen. For example, the transient and period of $(1,1,1,1,1)$ are both 3:

	0	1	1	1	1	1
	1	5				
	2	1	4			
transient	3	2	3			
	4	1	2	2		
	5	1	1	3		
period	6	2	3			

Here is a plot of the transients and periods of all starting lists $A = (n)$ for $n \leq 50$.



Note the fixed points $D(A) = A$, the few red dots at the bottom.

Task

A. Show that all transients must be finite.

B. Characterize all the fixed points of the DASZ operation.

C. Determine which initial lists $A = (n)$ lead to a fixed point.

Solution: DASZ Operator

Part A: Repeat

The key insight is that for any list $L = (a_1, a_2, \ldots, a_w)$ the application of D does not affect the weight of L, defined as $w(L) = \sum_i a_i$. Hence, the weight is an invariant with respect to our operation. Since the entries a_i are non-negative there are only finitely many lists of a given weight, hence repeated application of D must ultimately result in a cycle: $D^{t+p}(L) = D^t(L)$ for some $t \geq 0$, $p > 0$ (the transient and period).

Part B: Fixed Points

Consider a list $L = (a_1, a_2, \ldots, a_n)$ assumed to be sorted in non-decreasing order. Suppose L is a fixed point. Since the length of $D(L)$ is $n - 1 + k$ where k is maximal such that $a_k = 1$ we must have $1 = a_1 < a_2$. An easy induction then shows that $a_i = i$. It is clear that all lists $(1, 2, 3, \ldots, n-1, n)$ are fixed points, done.

Part C: To FPs

Since application of D does not affect weight, a fixed point of width m must have weight $w = m(m+1)/2 = T_m$, the mth triangular number, for $m > 0$. Hence we only have to consider numbers $1, 3, 6, 10, 15, 21, \ldots$ as starting points (the red dots at the bottom in the picture). A little experimentation leads to the following:

Claim: All the lists (T_m) evolve to their corresponding fixed points $(1, 2, \ldots, m)$ in T_{m-1} steps.

This is intuitively clear from a table describing the orbit of, say, $L_6 = (21)$.

0	21					
1	1	20				
2	2	19				
3	1	2	18			
4	1	3	17			
5	2	3	16			
6	1	2	3	15		
7	1	2	4	14		
8	1	3	4	13		
9	2	3	4	12		
10	1	2	3	4	11	
11	1	2	3	5	10	
12	1	2	4	5	9	
13	1	3	4	5	8	
14	2	3	4	5	7	
15	1	2	3	4	5	6

For an actual proof start with a warm-up exercise: let's consider lists of the form $L_k = (1, 2, \ldots, k, \infty)$ where ∞ stands for a very large number, assuming $\infty - 1 = \infty$.

Claim 1: L_k evolves to L_{k+1} in $k + 1$ steps.

To see this, show by induction on $0 \leq s \leq k$ that

$$D^s(L_k) = (1 + \delta_{1,s}, 2 + \delta_{2,s}, \ldots, k + \delta_{k,s}, \infty)$$

where $\delta_{i,s} = 0$ if $i + s \leq k$ and 1 otherwise. Hence

$$D^k(L_k) = (2, 3, \ldots, k, k+1, \infty)$$

and in one more step we get L_{k+1}.

Note that ∞ can be replaced by any number larger than all the other list elements that occur in the orbit of L_k. We write $L_k(x)$ for the list obtained by replacing ∞ by x in L_k. It is immediate from claim 1 that $L_k(x)$ evolves to $L_{k+1}(x - k - 1)$ in $k + 1$ steps. A simple induction using claim 1 then shows that

Claim 2: $L_0(T_m)$ evolves to $L_k(T_m - T_k)$ in T_k steps.

But then the main claim follows: $L_0(T_m)$ is none other than the initial configuration (T_m).

3. Speeding Up Iteration (30)

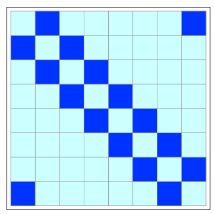
Background

The method of fast exponentiation can sometimes be used to speed-up the computation of $f^t(a)$ for some endofunction $f : A \to A$. Here is an example, and a limitation to this speed-up effect.

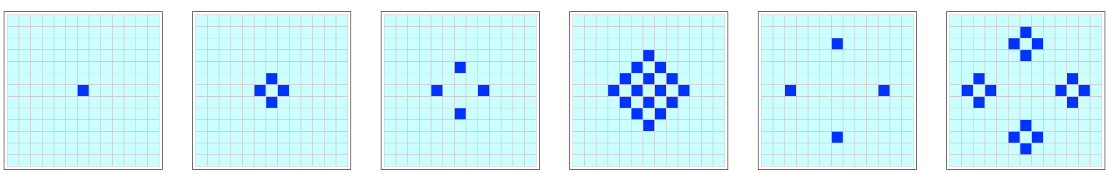
For $n \geq 1$ let $A = \mathbf{2}^{n \times n}$ be the set of all $n \times n$ Boolean matrices. Define the circulant matrix C by

$$C(i,j) = \begin{cases} 1 & \text{if } j = i \pm 1 \\ 0 & \text{otherwise.} \end{cases}$$

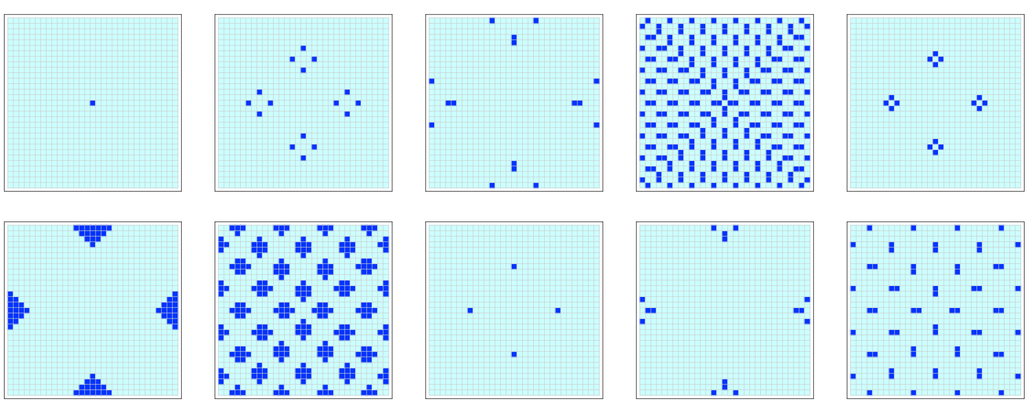
Here the indices are supposed to wrap around, so that, say, C_8 has the form



Lastly, define $f : A \to A$ by $f(X) = C \cdot X + X \cdot C$ where for the matrix multiplication we interpret addition as logical *exclusive or* and multiplication as logical *and*. Here is the effect of applying f^t to the 13×13 matrix with a single 1 in the center, rest all 0's, for $t = 0, 1, \ldots, 5$.



Note how, at times 2 and 3, 4 and 5, the pictures contain 4 copies of the pictures at times 0 and 1. Similarly, the effect of f^t on the 31×31 single-point matrix, for times $t = 0, 10, 20, \ldots, 90$.



The patterns are rather surprising, you might want to write a program that the produces the whole orbit (and try different matrix sizes).

Task

A. Describe the effect of f on $X \in A$ in geometric terms.

B. Show how to compute $f^t(X)$ for $X \in A$ in time $O(\mathsf{pol}(n) \log t)$ where pol is a low-degree polynomial depending only on n. Make sure to explain the degree of pol.

   Hint: express f as a single matrix multiplication. You might want to look up Kronecker product.

C. Show that $\mathbb{P} = \mathbb{NP}$ if exponential speed-up is always possible.

Comment

For part (C), find a way to determine satisfiability of a Boolen formula $\phi(x_1, \ldots, x_n)$ by iterating a function f defined essentially on $\mathbf{2}^n$.

Solution: Speeding Up Iteration

Part A: Xor

Note that the effect of $X \cdot C$ is to rotate the rows of X left and right, and $C \cdot X$ similarly rotates the columns. So a single bit spreads out to its four neighbors (things wrap around, we are dealing with a torus rather than a square).

Since we are using logical $\oplus$ and $\wedge$, the algebra takes place in $\mathbb{Z}/(2)$, the two-element field; A is a vector space of dimension n^2 over this field, and f is a linear map. Thus, f can be represented by a $n^2 \times n^2$ matrix M: $f(X) = M \cdot X$ (think of X as a column vector).

But then we can use fast exponentiation to compute M^t in $\emptyset(\log t)$ matrix multiplications. A single one of these multiplications is $\emptyset(n^6)$ using brute force (though speedups are possible using fast matrix multiplication).

Incidentally, there is another way to tackle this problem: try find something like a closed form solution to the problem of computing the bit $f^t(X)(i, j)$. This involves quite a bit of messy algebra involving binomials, but can also be used to speed-up the computation.

Part B: No Speed-Up

Define $A = \mathbf{2}^n \cup \{\perp\}$ where $\perp$ is some new element. Let $\varphi(x_1, \ldots, x_n)$ be a Boolean formula and define $f : A \to A$ as follows: $f(\perp) = \perp$ and

$$f(\boldsymbol{x}) = \begin{cases} \perp & \text{if } \boldsymbol{x} \text{ satisfies } \varphi \\ \boldsymbol{x} + 1 & \text{otherwise.} \end{cases}$$

Here $\boldsymbol{x} + 1$ is meant as: increment the corresponding n-bit number in binary.

But then f is polynomial time computable and φ is satisfiable iff $f^{2^n}(\mathbf{0}) = \perp$. Speed-up would get us down to $\emptyset(\mathsf{pol}(n)n)$, collapsing $\mathbb{NP}$ to $\mathbb{P}$.