# Relations Between the Low Subrecursion Classes

**Cristian Grozea**
Faculty of Mathematics, Bucharest
University, Bucharest, Romania

CDMTCS-158
July 2001

# RELATIONS BETWEEN THE LOW SUBRECURSION CLASSES

CRISTIAN GROZEA [*]

**Abstract**

Much work has been done in order to get a single, natural hierarchy of the primitive recursive classes. The most famous hierarchy, Grzegorczyk's $(\mathcal{E}_r)_{r\in\mathbf{N}}$, has problems at lower levels, as it is not useful in characterizing the polynomial time computable functions (probably the most important class of computable functions, the class of feasible functions). Therefore, several other hierarchies have been proposed, culminating with the most recent results of S.J. Bellantoni, K.H. Niggl [1], where an interesting ranking of the primitive functions is described, leading to two hierarchies, $(\mathcal{PR}_1^r)_{r\in\mathbf{N}}$ (defined by means of primitive recursion) and $(\mathcal{PR}_2^r)_{r\in\mathbf{N}}$ (defined by replacing primitive recursion with recursion on notation - binary recursion - in the definition of $\mathcal{PR}_1$). As for $\forall r \geq 2, \mathcal{PR}_1^r = \mathcal{PR}_2^r = \mathcal{E}_{r+1}$, $\mathcal{PR}_1^1 = \mathcal{E}_2$, and $\mathcal{PR}_2^1 = P_f$, these two parallel hierarchies succeed to integrate both FLINSPACE (i.e. $\mathcal{E}_2$) and FPOLTIME (i.e. $P_f$) functions with the Grzegorczyk hierarchy, from the elementary level and above. It was already known that $P_f \not\subset \mathcal{E}_2$, and that $\mathcal{E}_2 \not\subset P_f$, provided $P \neq NP$ (R.V. Book [2])

In this paper we prove that the above results cannot be improved assuming $P \neq NP$, i.e. there is no unique hierarchy to contain both $\mathcal{E}_2$ (or $\mathcal{E}_1$, or $\mathcal{E}_0$) and $P_f$, as $P_f$ is not somewhere in between $\mathcal{E}_0$ and $\mathcal{E}_3$. Although both $\mathcal{E}_0$ and $P_f$ are subsets of $\mathcal{E}_3$, we prove that $\mathcal{E}_0 \not\subset P_f$. In achieving this result we show that $SAT \in \mathcal{E}_0$ which is interesting in itself.

**Key words.** Grzegorczyk hierarchy, subrecursion classes, SAT, NP

**AMS subject classifications.** 03D20, 03D55

## 1 Introduction; Notation

We shall follow the notation in Rose [5] and denote by $\mathbf{q}(x, y)$ the arithmetical quotient of the integer division of $x$ by $y$, by $\mathbf{r}(x, y)$ the remainder of the integer division of $x$ by $y$, by $\mathbf{d}(x)$ the *length* of the binary representation of $x$ and by $\mathbf{D}(x, y) = 2^{\mathbf{d}(x)*\mathbf{d}(y)}$ the *smash* function.

[*]Faculty of Mathematics, Bucharest University, Str. Academiei 14, R-70109 Bucharest, Romania. Email: `chrisg@phobos.ro`

Let $(\mathcal{E}_r)_{r\in\mathbf{N}}$ be the Grzegorczyk hierarchy, and let $P_f$ be the set of the polynomial-time computable functions. We denote by $\mathcal{C}_*$ the subset of the Boolean-valued functions in the function set $\mathcal{C}$. Also $(P_f)_* = P$.

Note that $P_f$ contains the *smash* function $\mathbf{D}$, which is not in $\mathcal{E}_2$, as it is not polynomial bounded. Therefore $P_f \not\subset \mathcal{E}_2$.

It was already known that $\mathcal{E}_2 \not\subset P_f$, provided $P \neq NP$ (Book see [2], theorem 1).

We shall prove a much stronger assertion, namely

$$\mathcal{E}_0 \not\subset P_f, \text{ provided } P \neq NP.$$

This leads us to the conclusion that $\mathcal{E}_0$ and $P_f$ are not comparable and hence $P_f$ is not somewhere between $\mathcal{E}_0$ and $\mathcal{E}_3$.

In this paper, we show that the satisfiability problem of Cook (SAT), which is known to be $NP$-complete, can be solved in $\mathcal{E}_0$, so we will be able to explicitly show a function in $\mathcal{E}_0$, which is not in $P_f$.

## Recursion

The function $\mathbf{f}$ is defined by *primitive recursion* from $\mathbf{g}$ and $\mathbf{h}$ if $\mathbf{f}(0,\overrightarrow{y}) = \mathbf{g}(\overrightarrow{y})$ and $\mathbf{f}(x+1,\overrightarrow{y}) = \mathbf{h}(x,\overrightarrow{y}, f(x,\overrightarrow{y}))$.

The function $\mathbf{f}$ is defined by *limited recursion* from $\mathbf{g}, \mathbf{h}, \mathbf{F}$ if $\mathbf{f}$ is defined by primitive recursion from $\mathbf{g}$ and $\mathbf{h}$ and, additionally $\mathbf{f}(x,\overrightarrow{y}) \leq \mathbf{F}(x,\overrightarrow{y})$, for all $x$ and $\overrightarrow{y}$.

Let $\mathcal{E}_0$ be the class of (primitive recursive) functions whose initial functions are the zero function, the successor function, and is closed under composition and limited recursion. By $\mathcal{E}_1$ we denote the class of (primitive recursive) functions whose initial functions are the zero function, the successor function, the addition function $x+y$, and is closed under composition and limited recursion; $\mathcal{E}_1$ is the closure of $\mathcal{E}_0$ under addition. Formally, $\mathcal{E}_2$ is the class of (primitive recursive) functions whose initial functions are the zero function, the successor function, $x+y$ and $xy$, and is closed under composition and limited recursion; $\mathcal{E}_2$ is the closure of $\mathcal{E}_1$ under multiplication.

It is known that $\mathcal{E}_2$ consists of the linear-space computable functions (Ritchie [4], see also Rose [5]), and that $P_f$ has this recursive characterization (Cobham [3], see also Rose [5]): The polynomial-time computable functions are exactly those functions obtained from the initial functions 0, projections, the binary constructors ($\mathbf{s}_0(x) = 2*x$ and $\mathbf{s}_1(x) = 2*x+1$) and the *smash* function ($\mathbf{D}(x,y) = 2^{\mathbf{d}(x)*\mathbf{d}(y)}$), using composition and bounded recursion on notation (binary recursion).

## Logical operation and the least number operator $\mu$

We say that the predicate $\mathbf{P}$ is computed (represented) by the function $\mathbf{f}_P$ if $\mathbf{P}(x)$ is true if and only if $\mathbf{f}_P(x) = 0$.

We shall consider the (bounded) quantifiers $\forall t \leq x, \mathbf{P}(t)$ and $\exists t \leq x, \mathbf{P}(t)$ and the (bounded) least number operator $\mu$ :

$$\mu(t \leq x)[\mathbf{P}(x)] = \begin{cases} 0, & \text{if } \mathbf{P}(t) \neq 0, \forall t \leq x, \\ t_0, & \text{if } \mathbf{P}(t_0) = 0, t_0 \leq x, (\forall t < t_0, \mathbf{P}(t) \neq 0). \end{cases}$$

**Properties of low Grzegorczyk classes**

Here are some properties of low Grzegorczyk classes $\mathcal{E}_0$, $\mathcal{E}_1$, $\mathcal{E}_2$ (for proofs see Rose [5]):

(i) $x \dot{-} y \in \mathcal{E}_0$.

(ii) The logical operations can be represented in $\mathcal{E}_0$ (including the bounded quantifiers).

(iii) The least number operator $\mu$ can be defined in $\mathcal{E}_0$.

(iv) Each function in $\mathcal{E}_2$ has a polynomial upper bound.

(v) $\mathcal{E}_0 \subset \mathcal{E}_1 \subset \mathcal{E}_2$.

## 2 Low Grzegorczyk classes and $P_f$

The class $P_f$ contains the *smash* function $\mathbf{D}$, which is not in $\mathcal{E}_2$, as it is not bounded by polynomials. Therefore $P_f \not\subset \mathcal{E}_2$ and subsequently $P_f \not\subset \mathcal{E}_0$, $P_f \not\subset \mathcal{E}_1$.

We also show here that the other inclusions are false, that is :

$$\mathcal{E}_0 \not\subset P_f \tag{1}$$

(from this immediately follows that $\mathcal{E}_1 \not\subset P_f$ and $\mathcal{E}_2 \not\subset P_f$).

To prove (1) is enough to find a function that is not in $P_f$, but is in $\mathcal{E}_0$.

Let us consider the *satisfiability problem* (SAT), the problem that requires to decide for a given Boolean formula in CNF (conjunctive normal form) whether there exist Boolean values for the variables in the formula such that the formula evaluates to the value *true*. SAT is known to be $NP$-complete, so there is a function in $P_f$ for solving it if and only if $P = NP$.

We shall start here our construction proving that SAT can be solved by a function in $\mathcal{E}_0$. We will use *packed arrays* to encode the Boolean formula as a sequence of fixed-length binary representation symbols.

**Definition 1** *For a fixed symbol-length $k$, $(k \geq 1)$ a* **packed array** *is a natural number $\mathbf{a} \in \mathbf{N}$, such that the least significant $k$ bits of $\mathbf{a}$ codify the symbol at index 0, the next $k$ bits codify the symbol at index 1, and so on.*

**Example 1** *Let $a$ be the array containing the numbers 3, 7, 4, 5: $a[0] = 3$, $a[1] = 7$, $a[2] = 4$, $a[3] = 5$.*

*When we pack this array using 3 bits per symbol we get this natural number: 101100111011 = 101 100 111 011 (in binary notation, splited for more clarity; note the reverse order: the last group is the number $a[0] = 3$, the one before the last one is the number $a[1] = 7$ and so on).*

*When we pack this array using 4 bits per symbol we get this natural number: 0101010001110011 = 0101 0100 0111 0011 (in binary notation, splited for more clarity).*

*But this number could represent a different array when interpreted as being packed using 3 bits per symbol: 0101010001110011 = 0 101 010 001 110 011 which is the array containing the numbers 3, 6, 1, 2, 5.*

*The packed representation together with the choosen symbol length fully defines an array.*

A problem might arise when the number 0 is stored in the most significant bits. We shall avoid this problem by using 1-based codes for symbols. In this way, for a proper encoded array, it will be easy to determine the array length (using the function **ArrayLength**, defined below).

**Lemma 2** *The following functions are in $\mathcal{E}_0$:*

(i) *the length $\mathbf{d}(x)$, the remainder $\mathbf{r}(x, y)$.*

(ii) *arrays (indexed memory) access function* **GetAt**, *defined below.*

*Proof.* All the functions defined below are in $\mathcal{E}_0$ : $\mathbf{not}(x) = 1 \dot{-} x$, $\mathbf{le}(x, y) = \mathbf{not}(x \dot{-} y)$, $\mathbf{or}(x, y) = \mathbf{not}((1 \dot{-} x) \dot{-} y)$, $\mathbf{and}(x, y) = \mathbf{not}(\mathbf{or}(\mathbf{not}(x), \mathbf{not}(y)))$, $\mathbf{impl}(x, y) = \mathbf{or}(\mathbf{not}(x), y)$, $\mathbf{equal}(x, y) = \mathbf{and}(\mathbf{le}(x, y), \mathbf{le}(y, x))$, $\mathbf{diff}(x, y) = \mathbf{not}(\mathbf{equal}(x, y))$.

The *bounded if* function **bif** is defined as follows: if *bool* is true, then it returns *thenval* otherwise it returns *elseval*, provided both values are bounded by *bound* :

$$\begin{aligned}\mathbf{bif}(bool, thenval, elseval, bound) = \ & (\mu t \le bound)[\mathbf{and}(\\ & \mathbf{impl}(bool, \mathbf{equal}(t, thenval)),\\ & \mathbf{impl}(\mathbf{not}(bool), \mathbf{equal}(t, elseval))) = 1].\end{aligned}$$

The remainder function **r**:

$$\begin{cases} \mathbf{r}(0, y) = 0,\\ \mathbf{r}(x+1, y) = \mathbf{bif}(\mathbf{diff}(\mathbf{r}(x, y), y \dot{-} 1), \mathbf{r}(x, y) + 1, 0, y),\\ \mathbf{r}(x, y) \le x.\end{cases}$$

The characteristic function for addition:
$\mathbf{sum}(x, y, z) = \mathbf{and}(\mathbf{equal}(z \dot{-} x, y), \mathbf{equal}(z \dot{-} y, x))$.
The function **DIV2** divides $x$ by 2 (integer division):
$\mathbf{DIV2}(x) = (\mu t \le x)[\mathbf{or}(\mathbf{sum}(t, t, x), \mathbf{sum}(t, t, x \dot{-} 1)) = 1]$.
The function **SHR** shifts right a number $y$ by a number of positions $x$:

$$\begin{cases} \mathbf{SHR}(y, 0) = y,\\ \mathbf{SHR}(y, x+1) = \mathbf{DIV2}(\mathbf{SHR}(y, x)),\\ \mathbf{SHR}(y, x) \le y.\end{cases}$$

The length function: $\mathbf{d}(x) = (\mu t \le x)[\mathbf{SHR}(x, t) = 0]$.

Now we are ready to introduce the array functions:

$$\begin{cases} \mathbf{GetAt}(array, mask, 0) = \mathbf{r}(array, mask);\\ \mathbf{GetAt}(array, mask, pos+1) = \mathbf{GetAt}(\mathbf{SHR}(array, \mathbf{d}(mask) \dot{-} 1), mask, pos).\\ \mathbf{GetAt}(array, mask, pos) \le array.\end{cases}$$

You can see that a supplementary argument is used, $mask$. On call, $mask = 2^k$, where $k$ is the symbol length, expressed in bits. Note that we get the mask as an input, we are not computing it.

The pair $(array, mask)$ completely defines an array in this representation.

This function returns the length of an array:
$\mathbf{ArrayLength}(array, mask) = (\mu t \leq array)[\mathbf{GetAt}(array, mask, t) = 0]$.

This function can be used to access the bits of the number $word$:
$\mathbf{GetBit}(word, pos) = \mathbf{GetAt}(word, 2, pos)$. □

We need a good encoding in order to ensure the polynomial size encoding of the Boolean formula (polynomial in what it is considered to be the size of the Boolean formula). Otherwise, for some disastrous representation of the formula, it might be possible for a polynomial time algorithm to solve the satisfiability problem (SAT), using the formula itself as a clock.

**Lemma 3** *The satisfiability problem (SAT) is solvable in $\mathcal{E}_0$.*

*Proof.* We will use the arrays defined above to construct a good representation for the Boolean formulas in CNF (conjunctive normal form).

We shall represent any CNF Boolean formula as a string of symbols, each symbol being:

(i) '+' (or), codified by 1, or

(ii) '*' (and), codified by 2, or

(iii) '-' (not), codified by 3, or

(iv) a variable index $i$, codified by the symbol $4 + i$.

We choose for the representation of the CNF Boolean formula the postfix (Polish) notation. So, for example, the formula $(x_0 + x_1) * (-x_2 + x_1)$ will be represented by the array containing these symbols: $4, 5, 1, 6, 3, 5, 1, 2$ and the natural number which represents this array for $k = 3$ (and therefore $mask = 8$) is 010001101011110001101100 (in binary notation).

In order to check whether a given Boolean CNF formula is satisfiable or not, we will evaluate the formula for all possible values of the variables. Each possible assignment of binary values to the variables can be represented by a natural number $assignment \leq 2^n - 1$, whose bits are the values of the $n$ variables. Note that we shall not compute this value $(2^n - 1)$, it will be given as part of the description of the formula (see the discussion below).

We shall use below the function:
$\mathbf{EvalVar}(index, assignment) = \mathbf{GetBit}(assignment, index)$.

If we have the formula, and the maximum assignment to be tested, simply test all the assignments, evaluating the Boolean formula for each assignment. If any of the assignments validates the formula, then the formula is satisfiable, otherwise it is not validable.

This is the function for deciding SAT:

$$\begin{aligned}&\mathbf{Validable}(formula, mask, maxassignment)\\&= (\exists\ assignment \le maxassignment)(\mathbf{Eval}(formula, mask, assignment) = 1).\end{aligned} \tag{2}$$

The function must be called with $maxassignment = 2^n - 1$, where $n$ is the number of variables in the formula.

We shall work under the reasonable assumption that each variable is represented (has at least one occurrence) in the formula. Note that any SAT instance can be converted to this form in polynomial time, so this form is not weaker than the general case. For the instances of SAT of this form, we can simply take $maxassignment = formula$. Some more work is done by the **Validable** function, but we do not care about time complexity in $\mathcal{E}_0$.

We do, however, care about time complexity when we are in $P_f$, and we claim that for the representation of Boolean formulas as explained above, no polynomial time algorithm can evaluate the function **Validable** given by the equation (2).

In order to complete the proof of this lemma, we must give the expression of the function **Eval**, used in the equation (2) for evaluating the Boolean formula in a given assignment.

This single loop program below evaluates a formula, represented as an array, in a Polish notation, given some assignment of values to variables:

```
//inputs: formula, mask, assignment
conjvalue = 1;
disjvalue = 0;
varvalue = 0;
for(i = 0;i<ArrayLength(formula,mask);i++)
{
if(GetAt(formula,mask,i) = '-')
then varvalue = NOT varvalue;
else if(GetAt(formula,mask,i) = '+')
then disjvalue = disjvalue OR varvalue;
else if(GetAt(formula,mask,i) = '*')
then conjvalue = conjvalue AND disjvalue;disjvalue = 0;
else varvalue = EvalVar(GetAt(formula,mask,i)-4, assignment);
}
return conjvalue;
```

Let the function computed by the program above be

$$\mathbf{Eval}(formula, mask, assignment)$$

We will prove that this function is in $\mathcal{E}_0$.

We start by defining several auxiliary functions, every one of which is in $\mathcal{E}_0$, being defined from functions already in $\mathcal{E}_0$ and using only constructions allowed in $\mathcal{E}_0$, such as the least number operator $\mu$, the composition and the bounded recursion.

The function **Encode** packs three bits $x$, $y$ and $z$ into a natural number $zyx$ between 0 and 7.

$$\begin{aligned}\mathbf{Encode}(x,y,z) = \quad & (\mu t \leq 7)[\mathbf{and}(\mathbf{equal}(\mathbf{GetBit}(t,0),x), \\ & \mathbf{and}(\mathbf{equal}(\mathbf{GetBit}(t,1),y),\mathbf{equal}(\mathbf{GetBit}(t,2),z))) = 1]\end{aligned}$$

The next three functions unpack a natural number between 0 and 7 in its three component bits.

```
GetVarVal(encvar) = GetBit(encvar,0)

GetDisVal(encvar) = GetBit(encvar,1)

GetConVal(encvar) = GetBit(encvar,2)
```

The function **EvalVar** returns the value of one of the formula's variables in the given assignment (environment).

```
EvalVar(index,assignment) = GetBit(assignment,index)
```

The function **if**:

```
if(bool,thenval,elseval) = bif(bool,thenval,elseval,1)
```

The next three functions compute the new values of the variables *varvalue*, *conjvalue*, *disjvalue* of the program above, after executing one iteration of the program.

```
VarVal(encvar,sym,assignment) =
if(le(sym,3),
if(equal(sym,3), not( GetVarVal(encvar))
 , GetVarVal(encvar))
,EvalVar(sym-4,assignment))

ConVal(encvar,sym) =
if(equal(sym,1),
and(GetConVal(encvar),GetDisVal(encvar))
,GetConVal(encvar))

DisVal(encvar,sym) =
if(diff(sym,1),
if(equal(sym,2),
or(GetVarVal(encvar),GetDisVal(encvar))
,GetDisVal(encvar))
,0)
```

This function corresponds to the execution of one iteration step of the program above, in the presence of the curent formula symbol *sym*, which can be either a variable or an operator code.

```
IterationStep(encvar,sym,assignment) =
Encode(
VarVal(encvar,sym,assignment)
,DisVal(encvar,sym)
,ConVal(encvar,sym))
```

The function **EvalAux**$(formula, mask, assignment, n)$ runs the above iteration $n$ steps and returns the packed number containing the values of the program variables *varvalue*, *disjvalue*, *conjvalue*. The first rule establish the initial values of those variables.

```
EvalAux(formula,mask,assignment,0) = Encode(0,0,1)

EvalAux(formula,mask,assignment,i+1) =
IterationStep(EvalAux(formula,mask,assignment,i),
GetAt(formula, mask,i), assignment)
```

Note that **EvalAux**(...) is bounded by the constant function 7, so it is defined by bounded recursion.

Finally, here is the definition of the function **Eval** that mimics the whole program behavior; it simply runs the iteration for each symbol of the formula to be evaluated and returns in the final the value of the variable *conjvalue*.

```
Eval(formula, mask, assignment) =
GetConVal(EvalAux(formula, mask, assignment,
ArrayLength( formula, mask )))
```

□

**Theorem 4** *Provided $P \neq NP$, we have, for $i = 0, 1, 2$:*

$$\mathcal{E}_i \not\subset P_f,$$

$$\mathcal{E}_i \text{ and } P_f \text{ are not comparable,}$$

$$\mathcal{E}_{i*} \not\subset P.$$

*Proof.* If $P \neq NP$, the function **Validable** defined in Lemma (2) cannot be in $P_f$, while $Validable \in \mathcal{E}_0$. The key point is that the value *maxassignment* it needs to decide SAT has polynomial length and our Boolean formula has polynomial length representation.

So we have explicitly constructed a function that is in $\mathcal{E}_0$, but not in $P_f$. Actually, the function **Validable** is binary valued, so it is in $\mathcal{E}_{0*}$ but not in $P$.

This result leads us to the conclusion that $\mathcal{E}_0$ and $P_f$ are not comparable (with respect to set theoretic inclusion).

The last relations for $i = 1, 2$ are direct consequences of the following set theoretic inclusions (see Rose [5]) which hold true for all $r \geq 0$: $\mathcal{E}_{r*} \subset \mathcal{E}_r$, $\mathcal{E}_r \subset \mathcal{E}_{r+1}$, $\mathcal{E}_{r*} \subseteq \mathcal{E}_{r+1*}$.

Therefore $\mathcal{E}_1$ and $P_f$ are not comparable either; same holds for the pair $\mathcal{E}_2$ and $P_f$. □

## 3 Conclusion

We have proved that $P_f$ is not somewhere between $\mathcal{E}_0$ and $\mathcal{E}_3$. As a consequence there can not exist a hierarchy of primitive recursive functions that contains as levels $P_f$ and $\mathcal{E}_3$ and any of $\mathcal{E}_0$, $\mathcal{E}_1$, $\mathcal{E}_2$. Indeed, the presence of $P_f$ in any hierarchy prevents the occurence of $\mathcal{E}_0$, $\mathcal{E}_1$ and $\mathcal{E}_2$.

## 4 Future research. The relation between $NP$ and $\mathcal{E}_0$

We shall show here the sketch of the proof that another NP-complete decision problem, the Hamiltonian cycle problem is also in $\mathcal{E}_0$.

**Question 5** *Is the whole $NP$ predicates class included in $\mathcal{E}_{0*}$?*

This question is open for the moment.

We assume that the Hamiltonian cycle problem is known to the reader.

In order to derive a solution to this problem in $\mathcal{E}_0$ we shall choose first a convenient encoding for the instances of this problem.

We shall rely on all the functions constructed for the $SAT$ problem.

Let $G$ be the non-directed graph, $n$ the vertexes count, $E$ the edges set.

We shall represent each number between 1 and $n$ on $k$ bits (pick the smallest $k$ usable). As before, there is an associated number $mask = 2^{k-1}$.

What should be mentioned is that in $\mathcal{E}_0$ many things can be done, as long as a big number is provided.

This big number we shall note here $BN$ and it must have this property: it must be greater than any number that occurs in the computations below.

We shall use for passing the edges of the graph the adiacency matrix $A$. We shall encode this matrix as a packed array with element size of $k$ bits, this way: $A[\mathbf{conc}(i, j, mask, BN)] = a_{i,j}$, for all $i, j = 1, 2, \ldots n$, where $a_{i,j} = 1$ iff $(i, j) \in E(G)$ and $a_{i,j} = 2$ otherwise. Please note that we are using the values 1 and 2 instead of the usual 1 and 0.

The function **conc** concatenates the binary representations of the first two arguments:
$\mathbf{conc}(i, j, mask, BN) = \mu w \leq BN[GetAt(w, mask, 0) = j$
AND $GetAt(w, mask, 1) = i$
AND $(w$ SHR $(d(mask) \dot{-} 1))$ SHR $(d(mask) \dot{-} 1) = 0$
$]$.

The next function tests the edge $(i, j)$ existence.
$\mathbf{edge}(i, j, A, mask, BN) = GetAt(A, mask, \mathbf{conc}(i, j, mask, BN))$.

We shall encode a path (chain) as a sequence of $n$ numbers between 1 and $n$, and those sequences shall be encoded as packed arrays.

The next function tests if the path $L$ is really a valid path and a cycle in the graph.
**cycle**$(L, n, A, mask, BN) =$
$L \leq BN$
AND $\forall i \leq BN, 1 = impl(i > n, GetAt(L, mask, i) = 0)$
AND $\forall 1 \leq i \leq n, GetAt(L, mask, i) \geq 1$ AND $GetAt(L, mask, i) \leq n$
AND $\forall 1 \leq i \leq n \dot{-} 1, edge(GetAt(L, mask, i), GetAt(L, mask, s(i)), A, mask, BN) =$
$1$
AND $edge(GetAt(L, mask, n), GetAt(L, mask, 1), A, mask, BN) = 1$.

The next function tests if the path $L$ is a Hamiltonian cycle in the graph.
**hamiltcycle**$(L, n, A, mask, BN) =$ **cycle**$(L, n, A, mask, BN) =$
AND $\forall 1 \leq i \leq n, \exists 1 \leq p \leq n, GetAt(L, mask, p) = i$.

Finally, this is the function for testing if a given graph is Hamiltonian.
**hamiltgraphaux**$(n, A, mask, BN) =$
$\exists L \leq BN,$ **hamiltcycle**$(L, n, A, mask, BN)$.

And now the discussion about the big number $BN$. We observe that the packed array $A$ encoding the edges has at least $k * (n^2 - 1)$ bits and is bigger than any other value occuring in the computation of the functions above.
Therefore we can take $BN = A$:
**hamiltgraph**$(n, A, mask) =$ **hamiltgraphaux**$(n, A, mask, \mathbf{A})$
As this is a predicate and all the functions derived above, including **hamiltgraph**, are in $\mathcal{E}_0$, this concludes the proof that the Hamiltonian cycle problem can be solved in $\mathcal{E}_{0*}$.

Concerning the question if the whole $NP$ class is included in $\mathcal{E}_{0*}$, it must either be proved that in $\mathcal{E}_{0*}$ the polynomial reducibility can be implemented somehow or a counterexample must be found.

## Acknowledgments

The author thanks to Philip G. Drazin and H.E. Rose from Bristol University, UK and to C.S. Calude from Auckland University, NZ.

## References

[1] S.J. Bellantoni, K.H. Niggl, *Ranking primitive recursions: the low ɑrzegorczyk classes revisited*, SIAM Journal on Computing, volume 29, number 2, p.401-415, 1999.

[2] R.V. Book, *On languages accepted in polynomial time*, SIAM Journal on Computing, volume 1, p.281-287, 1972.

[3] A. Cobham, *The intrinsic computational difficulty of functions*, Logic, Methodology and Philosophy of Science, ed. Y. Bar-Hillel, North-Holland, p. 24-30, 1965.

[4] R.W. Ritchie, *Classes of Predictably Computable Functions*, Transactions of the American Mathematical Society, vol. 106, p.139-173, 1963

[5] H.E. Rose, *Subrecursion - Functions and Hierarchies,* Clarendon Press - Oxford, 1984.