Molecular Arithmetic Coding (MAC) Pseudo-Codes

Melih Şahin*

*Computer Engineering & Mathematics Student at Koç Univeristy. {melihsahin21}@ku.edu.tr

This work accompanies the research done by Melih Şahin, Beyza E. Ortlek, and Ozgur B. Akan.
Pdf of our paper can be found at: https://arxiv.org/abs/2403.04672

Algorithm 1 Zero-Order Arithmetic Coding for MC With EOF

Require: the bit-precision P, word to be encoded $word$, a function which maps each number k to the probability of the kth symbol, $prob$

▷ EOF symbol is included in $prob$

$whole = 2^P$

$N = prob.size$ ▷ N is the number of symbols

for i=1,..,N $c[i] = \sum_{k=1}^{i-1} prob[k]$

for i=1,..,N $d[i] = \sum_{k=1}^{i} prob[k]$

▷ determination of intervals for each symbol

$up\_is\_ONE = True$

$down\_is\_ZERO = True$

$s\_up = 1$

$s\_down = 0$

$s = 0$

$emit$ =[] ▷ $emit$ is initially an empty array

for $i = 1$ to $word.length$ **do**

  $the\_character = word[i]$

  $(emit\_new, s, a, b, down\_is\_ZERO, up\_is\_ONE, s\_down, s\_up, increase\_in\_s, pure\_increase)$ = **One_Time_Encoder**$(whole, c, d, prob, s, a, b, the\_character, down\_is\_ZERO, up\_is\_ONE, s\_up, s\_down)$

  $emit = emit + emit\_new$ ▷ + is the concatenation operator. Variables $increase\_in\_s$, $pure\_increase$ are redundant here and will be used in the decoding algorithm later.

end for

$emit = emit +$ **One_Time_Last_Character_Encoder_With_EOF**$(whole, s, a, b, down\_is\_ZERO, up\_is\_ONE, s\_up, s\_down)$

return emit

Algorithm 2 One Time Encoder

Require: $whole, c, d, prob, s, a, b, the\_character, down\_is\_ZERO, up\_is\_ONE, s\_up, s\_down$

$half = round(whole/2)$

$inverse\_golden\_ratio = 0.6180339887498948482$

$zero\_length = round(inverse\_golden\_ratio * whole)$

$one\_length = whole - zero\_length$

$Emit$ =[]

$increase\_in\_s = 0$

$pure\_increase = 0$

$character\_number = prob.index\_of(the\_character)$

$w = b - a$

$copy\_a = a$

$a = a + round((c[character\_number] * w)/whole)$

$b = copy\_a + round((d[character\_number] * w)/whole)$

if (not $down\_is\_ZERO$) and (not up_is_ONE) **then**

  if $b < one\_length$ or $a >= one\_length$ **then**

    if $b < one\_length$ **then**

      $a = round((a * (whole))/one\_length)$

      $b = round((b * (whole))/one\_length)$

      if $s > 0$ **then**

        $Emit.append(s\_down)$

        $new\_commute = 1$

        if $s\_down == 1$ **then**

          $new\_commute = 0$

        end if

        for $t = 0$ to $s - 2$ **do**

          if $t\%2 == 0$ **then**

            $Emit.append(new\_commute)$

          else

            $Emit.append(s\_down)$

          end if

        end for

      end if

      $Emit.append(10)$

      $pure\_increase += 2$

      $s = 0$

    else

      $a = round(((a - one\_length) * (zero\_length))/one\_length)$

      $b = round(((b - one\_length) * (zero\_length))/one\_length)$

      if $s > 0$ **then**

        $Emit.append(s\_up)$

        for $t = 0$ to $s - 2$ **do**

          $Emit.append(0)$

        end for

      end if

      $Emit.append(0)$

      $pure\_increase += 1$

    end if

    $down\_is\_ZERO = True$

    $up\_is\_ONE = True$

  end if

else if ($down\_is\_ZERO$) and (up_is_ONE) **then**

```
    ▷ we will not check if s > 0 as it is logically impossible
    while b < zero_length or a >= zero_length do
        if b < zero_length then
            Emit.append(0)
            pure_increase+ = 1
            a = round((a ∗ (whole))/zero_length)
            b = round((b ∗ (whole))/zero_length)
        else
            Emit.append(10)
            pure_increase+ = 1
            a = round(((a − zero_length) ∗ (whole))/one_length)
            b = round(((b − zero_length) ∗ (whole))/one_length)
        end if
    end while
else if (down_is_ZERO) and (not up_is_ONE) then
    if b < half or a >= half then
        if b < half then
            if s > 0 then
                Emit.append(s_down)
                new_commute = 1
                if s_down == 1 then
                    new_commute = 0
                end if
                for t = 0 to s − 2 do
                    if t%2 == 0 then
                        Emit.append(new_commute)
                    else
                        Emit.append(s_down)
                    end if
                end for
                s=0
            end if
            Emit.append(0)
            pure_increase+ = 1
            a = 2 ∗ a
            b = 2 ∗ b
        end if
        if a >= half then
            if s > 0 then
                Emit.append(s_up)
                for t = 0 to s − 2 do
                    Emit.append(0)
                end for
                s = 0
            end if
            Emit.append(0)
            pure_increase+ = 1
            a = 2 ∗ (a − half)
            b = 2 ∗ (b − half)
        end if
        down_is_ZERO = True
        up_is_ONE = True
    end if
end if
while True do
    if (down_is_ZERO) and (up_is_ONE) then
        special_number_0 = round((whole ∗ inverse_golden_ratio) ∗ inverse_golden_ratio)
        if a >= special_number_0 then
            a = round(((a − special_number_0) ∗ (whole))/(whole − special_number_0))
            b = round(((b − special_number_0) ∗ (whole))/(whole − special_number_0))
            ▷ this part [107-121] is explained in the Fig. 25
            if s == 0 then
                s_up = 1
                s_down = 0
            end if
            s+ = 1
            increase_in_s+ = 1
            down_is_ZERO = False
            up_is_ONE = False
        else
            break
        end if
    else if (not down_is_ZERO) and (not up_is_ONE) then
        special_number_1 = round((whole ∗ 2 ∗ inverse_golden_ratio) ∗ inverse_golden_ratio)
        if b < special_number_1 then
            a = round((a∗(whole))/special_number_1)
            b = round((b∗(whole))/special_number_1)
                ▷ this part is explained in the Fig. 12
            if s == 0 then
                s_up = 0
                s_down = 1
            end if
            s+ = 1
            increase_in_s+ = 1
            down_is_ZERO = True
            up_is_ONE = False
        else
            break
        end if
    else if (down_is_ZERO) and (not up_is_ONE) then
        special_number_2 = round((whole/2) ∗ (1 + inverse_golden_ratio))
        special_number_3 = round((whole/2) ∗ inverse_golden_ratio)
        if b < special_number_2 and a >= special_number_3 then
            a = round((a − special_number_3) ∗ 2)
            b = round((b − special_number_3) ∗ 2)
                ▷ this part is explained in the Fig. 3
            if s == 0 then
```

```
                s_up = 0
                s_down = 0
            end if
            s+ = 1
            increase_in_s+ = 1
            down_is_ZERO = False
            up_is_ONE = False
        else
            break
        end if
    end if
end while
return (Emit, s, a, b, down_is_ZERO, up_is_ONE,
        s_down, s_up, increase_in_s, pure_increase)
```

We will now be figuratively explaining interval rescaling section of Algorithm 2.

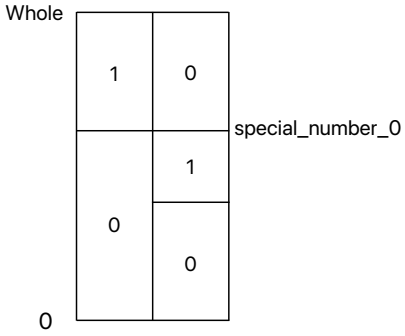

Fig. 1: Case 1

In the case where the current down and up bits are 0 and 1 respectively, if our interval [a,b) lies above the special_number_0 shown in Fig. 1, we are able to rescale our current interval such that the new intervals down bit is 1 and up bit is 0. The details are given in the Algorithm 2[107-121].

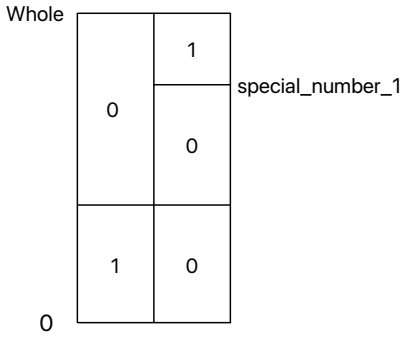

Fig. 2: Case 2

In the case where the current down and up bits are 1 and 0 respectively, if our interval [a,b) lies below the special_number_1 shown in Fig. 2, we are able to rescale our current interval such that the new interval's down and up bits are both 0. The details are given in the Algorithm 2[123-138].

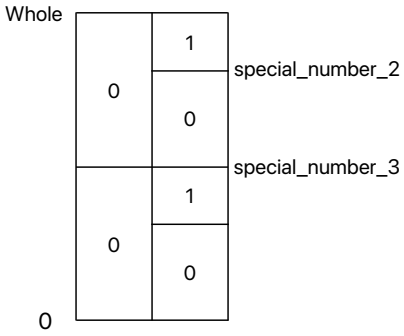

Fig. 3: Case 3

In the case where the current down and up bits are both 0, if our interval [a,b) lies in between special_number_3 and special_number_2 shown in Fig. 3, we are able to rescale our current interval such that the new intervals down bit is 1 and up bit is 0. The details are given in the Algorithm 2[139-156].

Algorithm 3 One Time Last Character Encoder With EOF

```
Require: whole, c, d, prob, s, a, b, down_is_ZERO,
    up_is_ONE, s_up, s_down
half = round(whole/2)
inverse_golden_ratio = 0.6180339887498948482
zero_length = round(inverse_golden_ratio * whole)
one_length = whole − zero_length
Emit =[]
while True do
    if s > 0 then
        if (not down_is_ZERO) and (not up_is_ONE)
         then
            if (a == 0 and b >= one_length) or
            (b == whole and a < one_length) then
                if b == whole and a < one_length then
                    Exact copy of Algorithm_6 [40-45]
                    break
                end if
                if a == 0 and b >= one_length then
                    Exact copy of Algorithm 2 [19-30]
                    break
                end if
            else
                down_bit_length = one_length − a
                upper_bit_length = b − one_length
                if down_bit_length >=
                upper_bit_length then
                    Exact copy of Algorithm 2 [19-30]
                    Emit.append(10)
                    b = whole
                    a = round((a * whole)/one_length)
                    if a >= zero_length then
                        Emit.append(10)
                        break
```

```
                end if
            else
                Exact copy of Algorithm 2 [40-45]
                a = 0
                b = round(((b − one_length) ∗ whole)/((whole − one_length)))
                if b < zero_length then
                    Emit.append(0)
                    break
                end if
            end if
            s = 0
            down_is_ZERO = True
            up_is_ONE = True
        end if
    else if (not down_is_ZERO) and (not up_is_ONE) then
        if (a == 0 and b >= half) or (b == whole and a < half) then
            if b == whole and a < half then
                Exact copy of Algorithm 2 [40-45]
                break
            else
                Exact copy of Algorithm 2 [19-30]
                Emit.append(0)
                break
            end if
        else
            down_bit_length = half − a
            upper_bit_length = b − half
            if down_bit_length >= upper_bit_length then
                Exact copy of Algorithm 2 [19-30]
                Emit.append(0)
                a = 2 ∗ a
                b = whole
            else
                Exact copy of Algorithm 2 [40-45]
                a = 0
                b = round((b − half) ∗ 2)
            end if
            s = 0
            down_is_ZERO = True
            up_is_ONE = True
        end if
    end if
else
    if ( down_is_ZERO) and (up_is_ONE) then
        if (a == 0 and b >= zero_length) or (b == whole and a < zero_length) then
            if b == whole and a < zero_length then
                Emit.append(10)
                break
            else
                Emit.append(0)
                break
            end if
        else
            down_bit_length = one_length − a
            upper_bit_length = b − one_length
            if down_bit_length >= upper_bit_length then
                Emit.append(0)
                b = whole
                a = round((a ∗ whole)/one_length)
            else
                Emit.append(10)
                a = 0
                b = round(((b − one_length) ∗ whole)/one_length)
            end if
        end if
    else if (not down_is_ZERO) and (not up_is_ONE) then
        if (a == 0 and b >= one_length) or (b == whole and a < one_length) then
            if b == whole and a < one_length then
                Emit.append(0)
                break
            else
                Emit.append(0)
                break
            end if
        else
            down_bit_length = one_length − a
            upper_bit_length = b − one_length
            if down_bit_length >= upper_bit_length then
                Emit.append(10)
                b = whole
                a = round((a ∗ whole)/zero_length)
            else
                Emit.append(0)
                a = 0
                b = round(((b − zero_length) ∗ whole)/zero_length)
            end if
            down_is_ZERO = True
            up_is_ONE = True
        end if
    else if (down_is_ZERO) and (not up_is_ONE) then
        if (a == 0 and b >= half) or (b == whole and a < half) then
            if a == 0 and b >= half then
                Emit.append(0)
                break
            else
                Emit.append(0)
                break
            end if
```

```
            else
                down_bit_length = one_length − a
                upper_bit_length = b − one_length
                if down_bit_length >=
                    upper_bit_length then
                    Emit.append(0)
                    a = 2 ∗ a
                    b = whole
                else
                    Emit.append(0)
                    a = 0
                    b = round((b − half) ∗ 2)
                end if
                down_is_ZERO = True
                up_is_ONE = True
            end if
        end if
    end if
end while
return (Emit)
```

Algorithm 4 Decoder With EOF

Require: $whole, bit\_sequence, c, d, alphabet, a, b,$ $down\_is\_ZERO\ (= True), up\_is\_ONE\ (= True),$ $symbol\_seqeunce, s\_up\ (= 1), s\_down\ (= 0), s\ (= 0)$

```
n = bit_seqeunce.length()
down_inter = Σ_{i=1}^{n} bit_sequence[i] ∗ round(whole ∗ (1/φ)^i)
if bit_sequence[n] == 0 then
    up_inter = down_inter + round(whole ∗ (1/φ)^n)
else if bit_sequence[n] == 1 then
    up_inter = down_inter + round(whole ∗ (1/φ)^{n−1})
end if
number = round((down_inter + up_inter)/2)
h= symbol ∈ alphabet such that number < d[symbol]
if h == NONE then return ERROR end if
symbol_sequence.append(h)
if h == EOF then return symbol_seqeunce end if
(emit_new, s, a, b, down_is_ZERO, up_is_ONE,
s_down, s_up, increase_in_s, pure_increase) =
One_Time_Encoder(whole, c, d, prob, s, a, b, h,
down_is_ZERO, up_is_ONE, s_up, s_down)
bit_sequence = bit_sequence[pure_increase +
increase_in_s :]
return Decoder_With_EOF(whole, bit_sequence, c, d
, alphabet, a, b, down_is_ZERO, up_is_ONE,
symbol_seqeunce, s_up, s_down, s)
```

Algorithm 5 Zero-Order Arithmetic Coding for MC Without EOF

Require: the bit-precision P, word to be encoded $word$, a function which maps each number k to the probability of the kth symbol, $prob$

```
Exact copy of Algorithm_5[1-15]
emit_last = One_Time_Last_Character_Encoder
_Without_EOF(whole, s, a, b, down_is_ZERO,
up_is_ONE, s_up, s_below)
return emit + emit_last
```

Algorithm 6 Decoder Without EOF

Require: $whole, bit\_sequence, c, d, alphabet, a, b,$ $down\_is\_ZERO\ (= True), up\_is\_ONE\ (= True),$ $symbol\_seqeunce, s\_up\ (= 1), s\_down\ (= 0), s\ (= 0)$

```
Exact copy of Algorithm_8[1-11, 13-14]
s_copy = s
emit_last = One_Time_Last_Character_Encoder
_Without_EOF(whole, s, a, b, down_is_ZERO,
up_is_ONE, s_up, s_below)
if (bit_sequence == emit_last[s_copy :]) then
    return symbol_sequence
end if
return Decoder_Without_EOF(whole, bit_sequence, c,
d, alphabet, a, b, down_is_ZERO, up_is_ONE,
symbol_seqeunce, s_up, s_down, s)
```

Algorithm 7 One Time Last Character Encoder Without EOF

Require: $whole, c, d, s, a, b, down\_is\_ZERO, up\_is\_ONE,$ s_up, s_down

```
half = round(whole/2)
number_of_added_bits = 0
inverse_golden_ratio = 0.6180339887498948482
zero_length = round(inverse_golden_ratio ∗ whole)
one_length = whole − zero_length
Emit =[]
while True do
    if (not down_is_ZERO) and (not up_is_ONE) then
        if a >= one_length then
            if s > 0 then
                Emit.append(s_up)
                for t = 0 to s − 2 do
                    Emit.append(0)
                end for
                s = 0
            end if
            Emit.append(0)
            number_of_added_bits+ = 1
            a = round(((a − one_length) ∗
            (zero_length))/one_length)
            b = round(((b − one_length) ∗
            (zero_length))/one_length)
            down_is_ZERO = True
            up_is_ONE = True
        else if b < one_length then
            Exact copy of Algorithm 2 [18-33]
            number_of_added_bits+ = 2
            a = round((a ∗ (whole))/one_length)
            b = round((b ∗ (whole))/one_length)
            down_is_ZERO = True
            up_is_ONE = True
        else
            Exact copy of Algorithm 2 [39-45]
```

$number\_of\_added\_bits+ = 1$
break
end if
else if ($down\_is\_ZERO$) and (not up_is_ONE) **then**
if $a >= half$ **then**
Exact copy of Algorithm 2 [39-45]
$a = 2 * (a - half)$
$b = 2 * (b - half)$
$down\_is\_ZERO = True$
$up\_is\_ONE = True$
else if $b < half$ **then**
Exact copy of Algorithm 2 [18-31]
$Emit.append(0)$
$a = 2 * a$
$b = 2 * b$
$down\_is\_ZERO = True$
$up\_is\_ONE = True$
else
Exact copy of Algorithm 2 [39-45]
break
end if
else if ($down\_is\_ZERO$) and (up_is_ONE) **then**
if $a >= zero\_length$ **then**
$Emit.append(10)$
$a = round(((a - zero\_length) * whole)/one\_length)$
$b = round(((b - zero\_length) * whole)/one\_length)$
else if $b < zero\_length$ **then**
$Emit.append(0)$
$a = round((a * whole)/zero\_length)$
$b = round((a * whole)/zero\_length)$
else
$Emit.append(10)$
break
end if
end if
end while
return $(Emit)$

Algorithm 8 Arithmetic Coder Suitability Checker With EOF

Require: the bit-precision P, word to be encoded $word$, a function which maps each number k to the probability of the kth symbol, $prob$

$Encoded\_word$=**Zero_Order_Arithmetic_Coding_for_MC_With_EOF**$(P, word, prob)$
$Decoded\_word\_1$=**Decoder_Without_EOF**$(P, Encoded\_word, prob)$
$Excess = [101010...10]$ ▷ the length of $Excess$ is P
$Decoded\_word\_2$=**Decoder_Without_EOF**$(P, Encoded\_word + Excess, prob)$
if ($Decoded\_word\_1$ is equal to $word$) and ($Decoded\_word\_2$ is equal to $word$) **then**
return True
else
return False
end if

Algorithm 9 Arithmetic Coder Suitability Checker Without EOF

Require: the bit-precision P, word to be encoded $word$, a function which maps each number k to the probability of the kth symbol, $prob$

$Encoded\_word$=**Zero_Order_Arithmetic_Coding_for_MC_Without_EOF**$(P, word, prob)$
$Decoded\_word$=**Decoder_Without_EOF**$(P, Encoded\_word, prob)$
if $Decoded\_word$ is equal to $word$ **then return True**
else return False end if