Tomasz Garbus 370795

1 Homework 1

Task 1: Consider multitape Turing machines in which before moving a head left, it is necessary to write a blank symbol. Do machines fulfilling this restriction recognize the same class of languages as standard Turing machines?
Let $\mathcal{M}$ be a regular single-tape Turing machine, i.e. $\mathcal{M} = (\Sigma_T, S, f, s_0, S_F)$, where:

- $\Sigma_T \supset \Sigma_I$ is the tape alphabet (Σ_I is input alphabet, $\perp \notin \Sigma_I$)
- S is the set of states
- $f : (S \times \Sigma_T) \to (S \times \Sigma_T \times \{-1, 0, 1\})$ is the transition function
- $s_0 \in S$ is the starting state
- $S_F \subset S$ is the set of accepting states

Let's construct a restricted (as in the task statement) Turing machine $\mathcal{M}^{(2)} = (\Sigma_T^{(2)}, S^{(2)}, f^{(2)}, s_0^{(2)}, S_F^{(2)})$. Let our machine $\mathcal{M}^{(2)}$ have two tapes, with the first tape being writable. The general concept is that we will split the input tape into two. The first tape will contain everything to the left of the head, the second – everything to the right of the head, in the reversed order.
Before simulating the original machine, we will perform a small preprocessing. We will move the head on the first tape to the end of the input and then back to the beginning, moving the symbols to the second tape.
Mostly $\mathcal{M}^{(2)}$ will be similar to the original machine.

- $\Sigma_T^{(2)} = \Sigma_T$
- $S^{(2)} = S \cup \{s_{mr}, s_{ml}\}$. State s_{mr} is dedicated to moving right in preprocessing phase, s_{ml} is dedicated to moving left in preprocessing phase. After the preprocessing phase, the machine will be in state s_0.
- $s_0^{(2)} = s_{mr}$
- $S_F^{(2)} = S_F$

Let $\perp$ be the blank symbol and $\triangleright$ the beginning of input. Let's now define the transition function $f^{(2)} : (S \times \Sigma_T^2) \to (S \times \Sigma_T^2 \times \{-1, 0, 1\}^2)$.
Below are the preprocessing steps:

$$f^{(2)}(s_{mr}, (\alpha, \perp)) = \begin{cases} (s_{mr}, (\alpha, \perp), (1, 0)) \texttt{ if } \alpha \neq \perp \\ (s_{ml}, (\perp, \perp), (-1, 1)) \texttt{ if } \alpha = \perp \end{cases}$$

(note that the second tape will always be empty in state s_{mr})

$$f^{(2)}(s_{ml},(\alpha,\perp)) = \begin{cases}(s_{ml},(\perp,\alpha),(-1,1)) \text{ if } \alpha \neq \triangleright \\ (s_0,(\alpha,\perp),(1,-1)) \text{ if } \alpha = \triangleright\end{cases}$$

As we can see, after the preprocessing we will end up in the state s_0, with blank first tape and the head right after the $\triangleright$ symbol, while the second tape will contain the reversed input and the head will be pointing to the first symbol from the input. Also, at all times, at most one head should be pointing to a non-blank symbol.

Below are the steps simulating the behavior of $\mathcal{M}$ with $\mathcal{M}^{(2)}$.

$$f^{(2)}(s \in S,(\alpha,\perp)) = \begin{cases}(s_1,(x_1,\perp),(1,-1)) \texttt{ if } k_1 = 1 \\ (s_1,(x_1,\perp),(0,0)) \texttt{ if } k_1 = 0 \\ (s_1,(\perp,x_1),(-1,1)) \texttt{ if } k_1 = -1\end{cases} \quad \text{where } f(s,\alpha) = s_1,x_1,k_1$$

$$f^{(2)}(s \in S,(\perp,\alpha)) = \begin{cases}(s_1,(x_1,\perp),(1,-1)) \texttt{ if } k_1 = 1 \\ (s_1,(x_1,\perp),(0,0)) \texttt{ if } k_1 = 0 \\ (s_1,(\perp,x_1),(-1,1)) \texttt{ if } k_1 = -1\end{cases} \quad \text{where } f(s,\alpha) = s_1,x_1,k_1.$$

So to conclude, we have constructed a "restricted" Turing machine $\mathcal{M}^{(2)}$ that recognizes the same language as $\mathcal{M}$.

Task 2: A finite set $C \subseteq \mathbb{N}^k$ (where $k \geqslant 2$) is called a *rectangle* if it can be represented as $C = A \times B$, for some $A \subseteq \mathbb{N}^i$ and $B \subseteq \mathbb{N}^j$, where $i + j = k$ and $i, j \geqslant 1$. We assume the standard representation of finite sets $C \subseteq \mathbb{N}^k$ in words over $\{0, 1, \$, \#\}$: a vector $\{v_1, ..., v_m\} \in \mathbb{N}^k$ is represented as $a_1\$a_2\$...\$a_k$ (where a_i is written in binary), and a set of vectors $\{v_1, ..., v_m\}$ is represented as $v_1\#v_2\#...\#v_m$ (notice that a set may allow multiple representations).
Prove that the set of words representing rectangles can be recognized by a deterministic Turing machine working in logarithmic space.
***Example*: the following two words should be accepted:**
`0$0#1$1#0$1#1$0` **and** `0$0$0#0$0$1#0$1011$0`**.**

First, we will verify that there is the same number of $ symbols between each pair of neighboring # symbols and also that there is at least one # symbol.

```
let w = input
let n = input length
let p = 0     # a helper counter to iterate through input word
let prev = -1 # length of the previous vector (or -1 if none)
let cur = 0   # length of the current vector
let vectors_count = 1
while p < n:
  if w[p] == '#':
    if prev != -1 and cur != prev:
      reject
    else:
```

```
      prev = cur
      cur = 0
    vectors_count = vectors_count + 1
  if w[p] == '$':
    cur = cur + 1
  p = p + 1
if cur != prev:
  reject      # there is no '#' after last vector so remember to check it
let k = cur   # k from the task description
if k < 2:
  reject
```

We will denote the x-th vector as `v_x`. Let's note that we can implement some useful procedues in a logarithmic space:

- `common_suffix(x, y, l)` – checks if v_x and v_y have a common suffix of length l. Otherwise, rejects.
- `common_prefix(x, y, l)` – checks if v_x and v_y have a common prefix of length l. Otherwise, rejects.

Now, we will look for satisfying values for i and j (as in task description) and for each pair, check whether for every vector prefix of length i occuring in our set and every vector suffix of length j occurring in our set, there is a vector being a concatenation of this prefix and suffix in our set.

```
i = 1
while i < k:
  let j = k - i  # takes log memory (we can always keep this
                 # counter on the same tape)
  let v1 = 1     # same goes for those 3 variables
  let v2 = 1
  let v3 = 1
  let found_pair = true   # this variable tells us whether the pair (i, j)
                          # is good (i.e. we have not verified that it is
                          # not satisfying)
  while v1 <= vectors_count:
    while v2 <= vectors_count:
      let exists = false        # this variable will tell us
                                # whether there is a vector which
                                # has a common prefix of length i
                                # with vector v1 and a common suffix
                                # of length j with vector v2
      while v3 <= vectors_count:
        if common_prefix(v1, v3, i) and common_suffix(v2, v3, j):
          exists = true
        v3 += 1
```

```
      if not exists:
        found_pair = false
      v2 += 1
    v1 += 1
  if found_pair:
    accept
# If we have not found a good pair (i, j) yet, we should reject.
reject
```

This roughly sketched algorithm for recognizing the language of rectangles can be implemented on a Turing machine using logarithmic space. Note that each variable used in the pseudocode is either boolean or logarithmic in size with regard to the length of the input word.
The functions `common_prefix` and `common_suffix` can be implemented in logarithmic memory as well. The general scheme would be as follows:

- keep a counter with already compared elements
- in order to navigate to one of the two compared vectors, keep track of the # symbols since the beginning of input word.
- in order to navigate to a concrete element, keep track of $ symbols
- perform a loop: navigate to one vector, read and remember one character, navigate to the other vector and compare the character, increase the counters