ADM 2019 - Assignment 1 : Compression

Meng Yao, Christiaan van Buchem, Auke Bruinsma

17 November 2019

This is the first assignment of the course Advanced Data Management for Data Analysis, 2019. In this report we will first describe the task. Then we will name the advantages and disadvantages of the 5 compression algorithms we wrote, this will help explaining our results. Next we will show you flowcharts which schematically show how the algorithms are implemented in the code. After that an instruction follows on how to compile and run our programs. Our compression rates will be presented in the results section and we will finish up the report with a discussion and conclusion section.

1 Assignment description

For this assignment we received multiple `.csv` files containing data of types `int` and `string`. The goal of this assignment is to:

1. Encode these `.csv` files according to 5 compression algorithms, which were discussed in the lectures. The encoded files need to be stored.
2. Decode the encoded files back to original form. These only need to be printed in the terminal.

2 Compression methods

The 5 compression algorithms are:

1. **Run-length encoding**: This algorithm stores sequences of the same data as a tuple which contains the value, the start position, and the run length. As you may expect, this kind of encoding is efficient for data where the same value occurs multiple times in a row. For example, if the letter 'w' occurs 10 times in a row from the second position in a file, they will be stored as (w, 1, 10).
2. **Bit-vector encoding**: For each unique value v in column c, a bit-vector b is created such that $b[i] = 1$ if $c[i] = v$. This means the number of bit-vectors that are created is equal to the number of unique elements in the array. So, this algorithm is most efficient on data which contains a small number of unique elements. .
3. **Dictonary encoding**: For each unique value a dictionary entry is created. This makes sure values that occur more than once are stored only one time. Then the file will be compressed as indice where each of them indicates which element in dictionary being shown here. Because of that, data that contains a lot of duplicate values are efficiently handled by this type of encoding.
4. **Frame of reference encoding**: This encoding algorithm works for integers only. The idea is to take a reference number which is close to all the elements in an array. The encoded array consists of the difference between the array values and the reference number. This encoding algorithm works efficient on an array containing large numbers which differ by a small amount. If the difference between the reference number and an array element is too large to be stored (a certain threshold can be set), the original number from the array is stored and an escape code, which indicates the difference is too large.
5. **Differential encoding**: This encoding mechanism is similar too frame of reference encoding. Instead of storing the difference w.r.t a reference number, the difference of two subsequent elements is stored. This algorithm is most efficient on, for example, time data, because this kind of data only increases by small amounts. Like frame of reference encoding, there's also a maximum number that can be stored. If the difference between two subsequent elements differs more than this threshold, the actual array value is stored. In a way, this is a reset value in the encoded array.

3 Implementation

In this section we present flowcharts for each of the 5 compression algorithms. These flowcharts show how we have implemented the algorithm and are a schematic overview of how our code functions. See the last 5 pages of this report for the figures.

4 How to compile & run our code

We chose to write in the language C++.All our encoding algorithms can be run from a single `.cc` file. This file needs to be compiled and run. In the program you need to specify (1) which you file you want to encode/decode, (2) if you want to en- or decode and (3) what kind of datatype you want to use[1]. If this is correctly specified, the program will run all 5 encoding or decoding algorithms on the `.csv` file.

5 Results

Table 1 specifies the size of the original files and the encoded files.

Table 1: This table contains the filename of the original `.csv` file and the original size in the two columns on the left. The five columns on the right give the encoded file size, for the encodings that are performed. Units are in MB's.

File	Size	`.bin`	`.rle`	`.dic`	`.for`	`.dif`
int files						
l_discount-int64.csv	12.5	46.9	65.0	12.5	15.3	16.4
l_discount-int8.csv	12.5	46.9	65.0	12.5	15.3	16.4
l_extendedprice-int32.csv	47.2	106.1				
l_extendedprice-int64.csv	47.2					
l_linenumber-int32.csv	12.0	46.9	68.4	12.0	13.9	13.3
l_linenumber-int8.csv	12.0	46.9	68.4	12.0	13.9	13.3
l_orderkey-int32.csv	46.9					
l_partkey-int32.csv	38.7					
l_quantity-int64.csv	16.9	46.9	74.3	16.8	24.0	25.4
l_quantity-int8.csv	16.9	46.9	74.3	16.8	24.0	25.4
l_suppkey-int16.csv	29.3					
l_suppkey-int32.csv	29.3					
l_tax-int64.csv	12.0	46.9	63.0	12.0	15.3	15.3
l_tax-int8.csv	12.0	46.9	63.0	12.0	15.3	15.3
string files						
l_comment-string.csv	165.0	-			-	-
l_commitdate-string.csv	66.0	-	123.4	27.3	-	-
l_linestatus-string.csv	12.0	-	10.0	12.0	-	-
l_receiptdate-string.csv	66.0	-	124.2	27.3	-	-
l_returnflag-string.csv	12.0	-	24.9	12.0	-	-
l_shipdate-string.csv	66.0	-	124.1	27.3	-	-
l_shipinstruct-string.csv	78.0	-	102.6	12.0	-	-
l_shipmode-string.csv	31.7	-	77.7	12.0	-	-

6 Discussion & Conclusion

In this section we will be discussing the results that we got from our code. First we will discuss the results for the `int` files and then we will look at the `string` files.

6.1 Implementation on integer files

For the integers it appears that we generally did not manage to have the compressed files be smaller than the original file. It was only for the `l_quantity-int64.csv` and its `int32` counterpart that

[1]In the script one needs to change the location of the data directory accordingly in `int main()`.

we managed to produce a smaller file using dictionary encoding. This comes as no surprise since these files contain a lot of data with but a small number of different data-entries.

The files compressed using bit-vector encoding We can also see that there is no difference in compression size for files of different `int` types. This is due to the fact that we were not able to alter the writing of the compressed files depending on the `int` type. Besides, another explanation of getting larger files is storing indice of the sparse matrix

For the `.for` and `.dif` algorithms, the compression size does not decrease for multiple files. This is the case because for these files the values are small and don't differ by a large amount. As specified in section 2, the `.for` algorithm is efficient on data with large values that are close together, and the `.dif` algorithm is efficient on large values where each subsequent element does not change a lot compared to the previous element. This is why for some files the file size does not decrease after compression.

6.2 Implementation on string files

According to the results on compressing string files, we can see that, on one hand, run length encoding ended up with larger files on 6 out of 7 tasks. It is because we need to save 3 entries at a time if lines in files don't repeat consecutively. On the other hand, dictionary encoding performed better than run length encoding. It reduced the size of files on 5 out of 7 tasks by a significant factor because of the small amount of unique values in the files and independence on the order of data. Combining with the results from integers files, where the size of compressed files are also low, dictionary encoding shows a low requirement on data types.

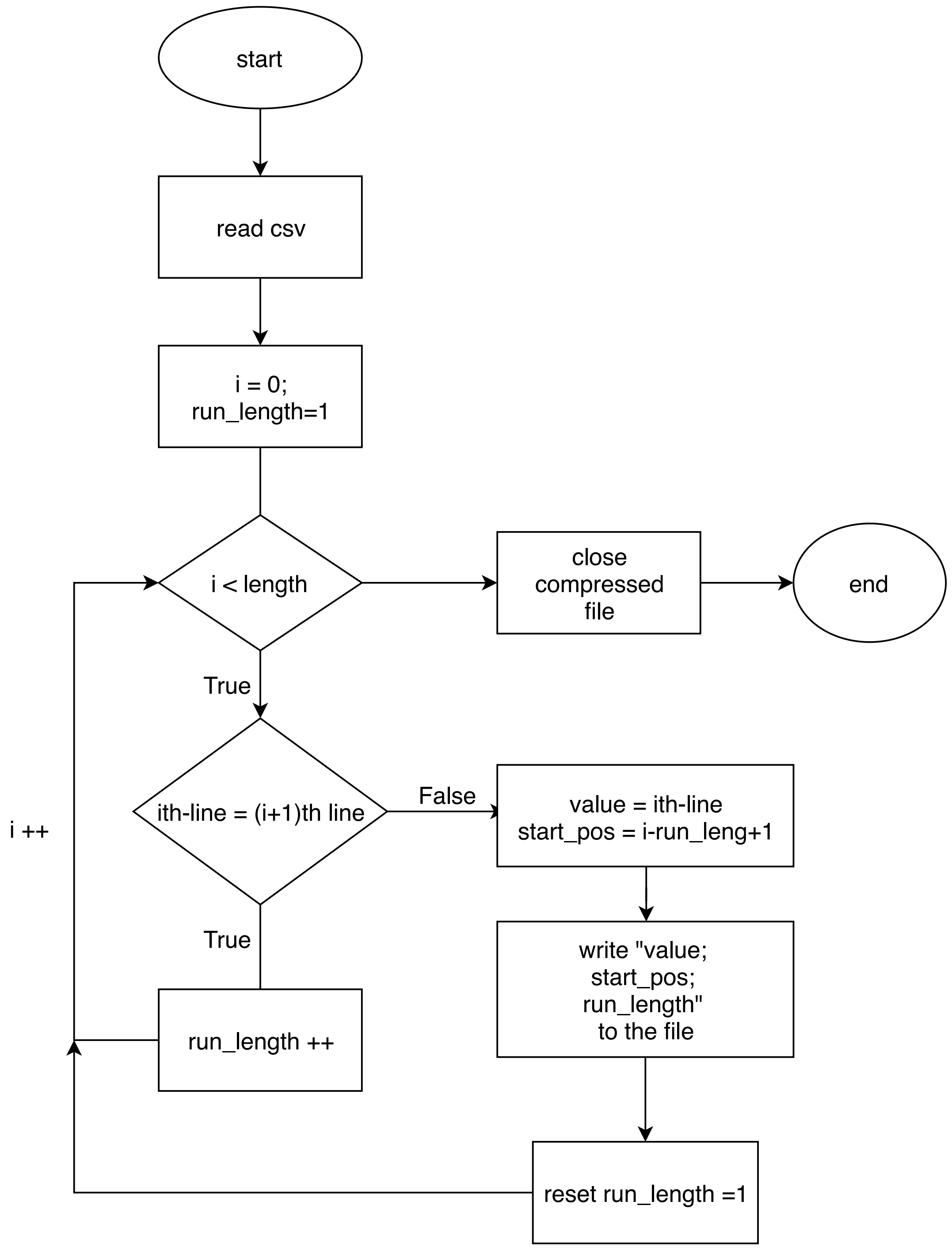

Figure 1: Flowchart of run-length encoding

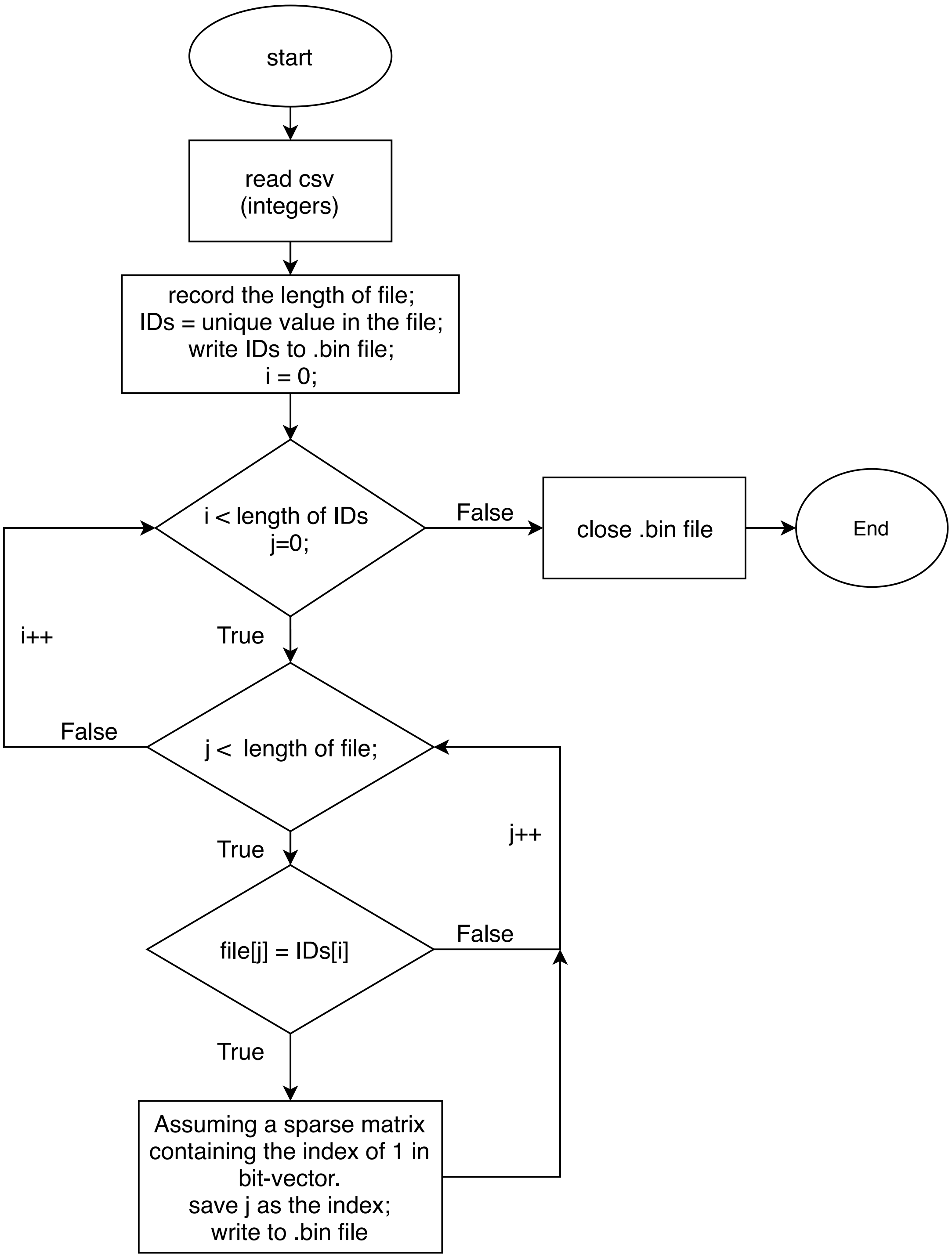

Figure 2: Flowchart of bit-vector encoding

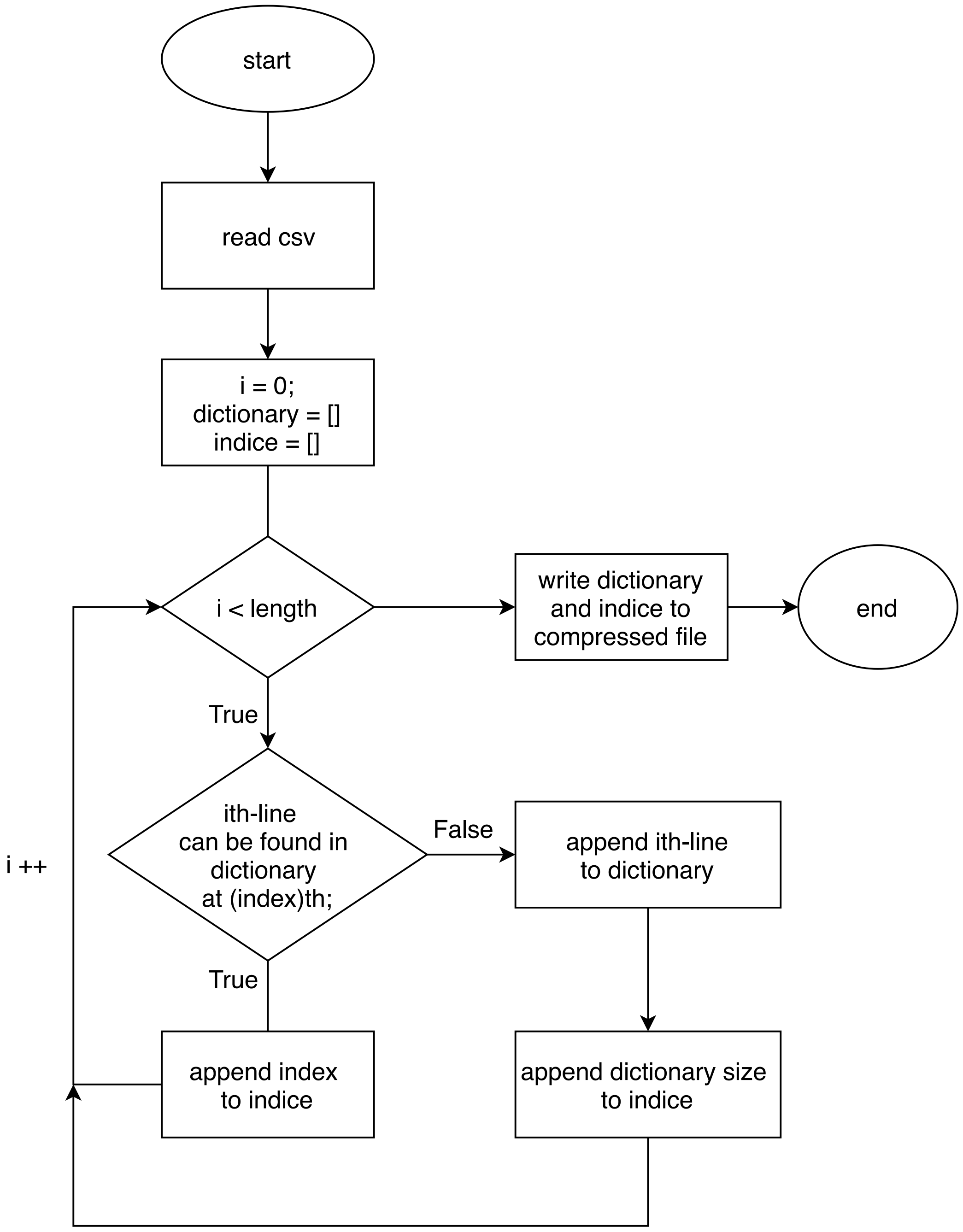
start
read csv
i = 0;
dictionary = []
indice = []
i < length
write dictionary and indice to compressed file
end
True
ith-line can be found in dictionary at (index)th;
False
append ith-line to dictionary
True
append index to indice
append dictionary size to indice
i ++

Figure 3: Flowchart of dictionary encoding

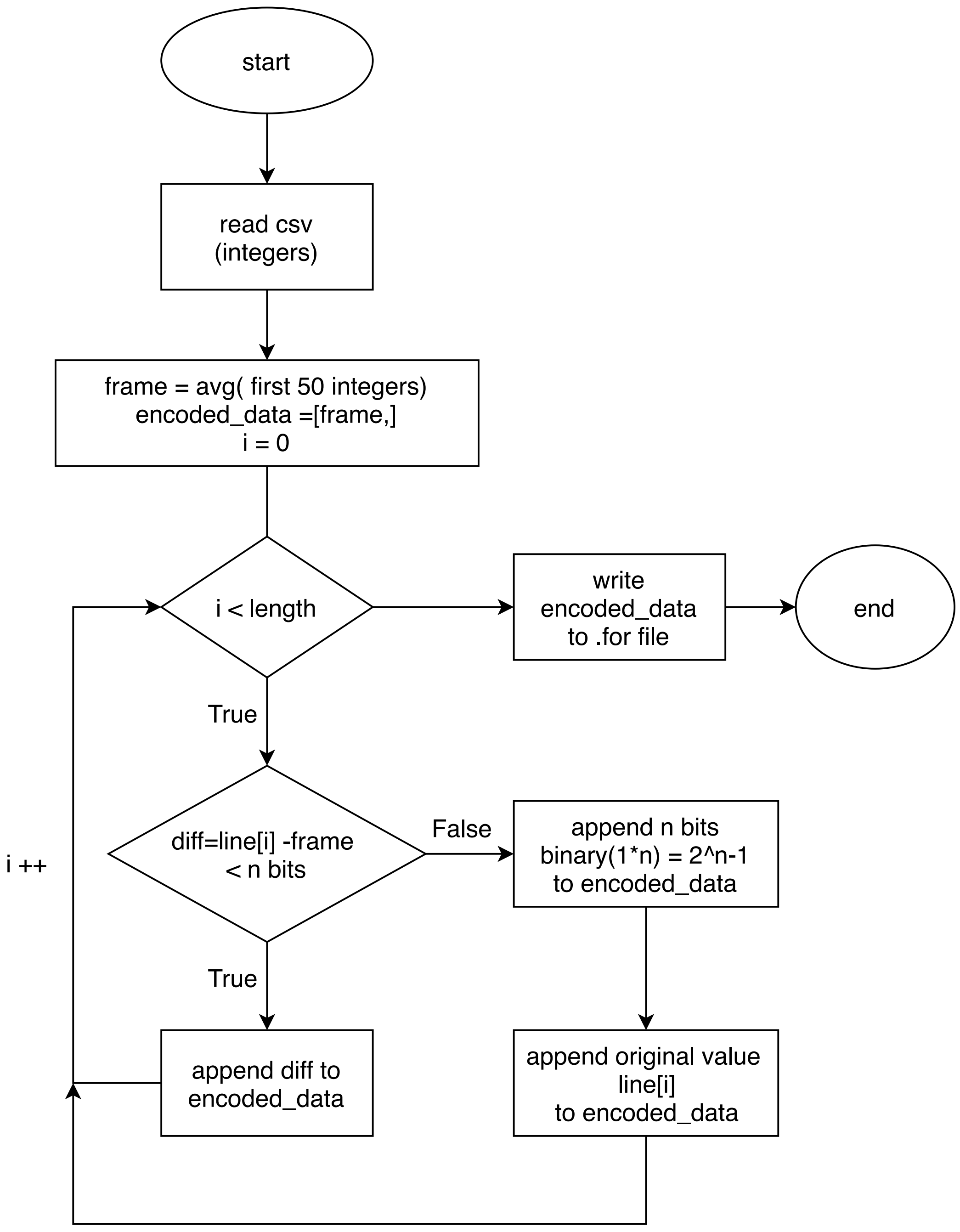

Figure 4: Flowchart of frame of reference encoding

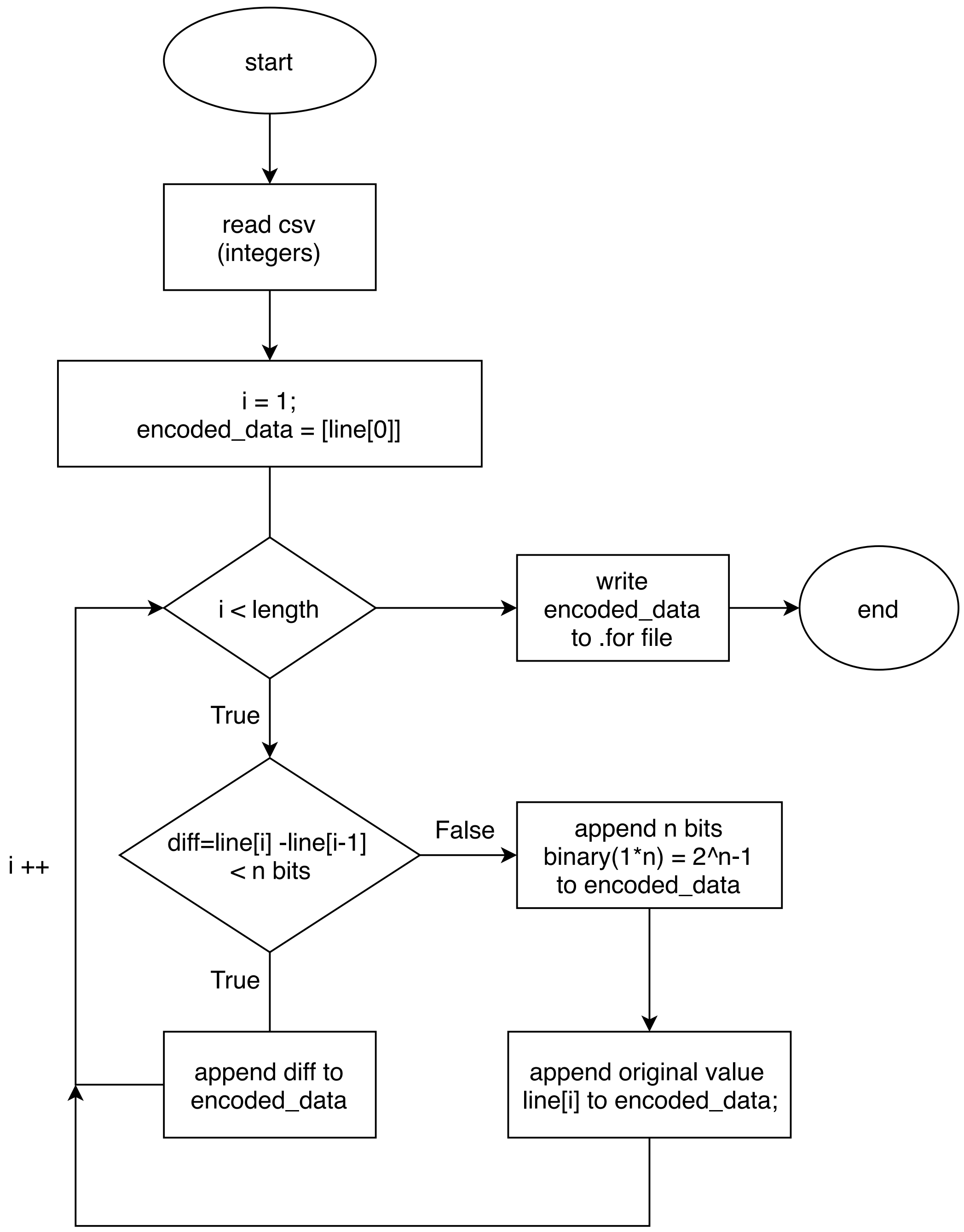
start
read csv (integers)
i = 1; encoded_data = [line[0]]
i < length
write encoded_data to .for file
end
True
diff=line[i] -line[i-1] < n bits
False
append n bits binary(1*n) = 2^n-1 to encoded_data
i ++
True
append diff to encoded_data
append original value line[i] to encoded_data;

Figure 5: Flowchart of differential encoding