CS214 Project 2023:
Image compression with automata

Prof L van Zijl and W van der Linden

February 2023

Change-log

Last update: 6 March 2023.

- Correction on example output for Figure 6 (Compression).
- Change in notation for Compression pseudo-algorithm, to use Article notations.
- Clarification on filepaths for input, and remote repository structure.
- Style update: 100 length lines, and TODO whitelisting up to MidTerm.
- Changes to Phase 3:
  - Discussion on the concepts of Lossy vs Lossless compression.
  - Removed mode 3: Black Quadrants. Mode 4 has been relabeled mode 3.
- Packaging requirements updated - treating the `src` folder as a source directory for your project means no java packaging is required. The specifications and scripts have been updated to reflect this.
- Correction in labels of example FA in Background section.

Due dates

Hand-in	Date	Time	Expected
Soft hand-in 1	3 Mar 2023	16h30	Git pipeline, script marking
Soft hand-in 2	31 Mar 2023	16h30	Student test suite phase 1
Phases 1 and 2	11 April 2023	13h30	Completed phases 1, 2
Phase 3	12 May 2023	13h30	Completed phase 3
Demo, remarks	15 and 19 May 2023	14h00	

Project brief

This project investigates image compression for images, based on automata theory. The method works particularly well for highly repetitive images (see Figure 1, for example). The idea is to encode the pixels of the image as strings from a given alphabet, and then store the finite automaton that recognizes all the words that represent strings in the image (more detail a bit later).

In this project, you are firstly required to recreate an image, given a specific finite automaton (the decompression part). Secondly, given an image, you are required to generate the finite automaton that will be able to reproduce the image (the compression part). Lastly, you will be required to add extra functionality to encode multi-dimensional images.

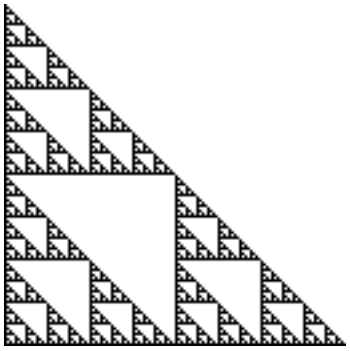

Figure 1: An image with a highly repetitive pattern

Rough marking scheme

- 75% – Test cases.
- 10% – Code style and Javadoc comments.
- 15% – General impression / bonus marks for effort exceeding the standard specification

The test cases are run for each phase, and the final mark for the test cases is calculated by the formula $0.3 * T_1 + 0.3 * T_2 + 0.4 * T_3$, where T_i represents the results for the test cases of phase i. The code style, comments, general impressions, and bonus marks are assigned once, after the hand-in of phase 3. Note that a mark of 100% is the maximum you can obtain, regardless of bonus marks awarded.
It is important that you use the correct style throughout the development of the project, instead of a mad rush effort before the final hand-in date.

Assessment

All marking will be done using scripts. It is therefore necessary that your project can be compiled and run from the command line. If it does not compile and run from the command line, you will receive zero for your project.

When marking the project, various aspects and components will be tested separately. As such, the project must be able to run in different modes.

Your project submission must include a `Compress.java` file, located in a `src` directory. **Use the default packaging approach**, i.e. no sub-directories or packages under `src`.
To compile your program, we will run

```
javac -d bin src/Compress.java
```

or

```
javac -d bin src/*.java
```

To run the program, we will always run

```
java -cp bin Compress gui mode multi-res [optional parameter] <input_file>
```

where the parameters are as described in the specification below, and the input file will contain a either an automaton, or an image.
Compilation and execution will happen relative to the root directory.

The output specification must be followed EXACTLY to facilitate script marking. Make sure that nothing except for the expected output is printed to standard output. Error message should be printed to the standard error stream.

A test script and some small examples will be provided through GitLab for every stage of the project, so that you can test the correctness of the code as you progress. We will run exactly the same script to mark your project, but with a more extensive test bank of input examples.

Hand-ins

You must hand in your project on git. The soft progress hand-ins occurs on Friday 3 March 2023 and Friday 31 March 2023, at 16h30. All hand-ins must conform to the style conventions. Feedback will be provided for each hand-in, so that you have time to fix any errors. Note that these soft hand-ins, if ignored, carry a penalty of -5% each, to be applied to your final project mark. That is, while you will receive no marks for a successful soft hand-in, you will receive the penalty if you do not hand in, or if the requirements of the hand-in is not met. For example, if the git pipeline shows style errors, or the script fails on your handin, you would get a -5% penalty.

The mid-semester hand-in is on Tuesday 11 April, at 13h30 (NOTE THE TIME), and the final hand-in on Friday 12 May, at 13h30. There is no remark of the mid-semester hand-in during the final hand-in. Please refer to the module framework for more detail.

You MUST follow the directory structure convention, for the hand-in. If you do not do that, the marking script will not work. You should not, under any circumstances whatsoever, use hard coded file paths in your code.

Specification

The project rests on some background theory, and an accompanied article. Make sure you understand these key concepts before you start work on the project.

Background

A finite automaton (FA) is a state-based model of computation. Much like any computer program, the FA can be in any one of a number of states. When an event occurs, the FA may move to another state. The FA starts in a designated start state, and ends in one of the designated final (also called accept) states. We model the events by specifying an alphabet, and let each event correspond to an alphabet symbol. If one then supplies a string of alphabet symbols, one can determine the path that the FA follows as it encounters those symbols in a sequential fashion. The set of all the input strings that cause the FA to end up in one of the final states, forms the language recognized by the FA.

For example, consider the simple FA in Fig. 2. This is an FA with eight states (q_0 to q_7). Its alphabet is the 26 lowercase letters `a` to `z`. Its start state is state q_0, and its (single) final state is state q_7. If one feeds the input string `cat` to the FA, it will accept it. Similarly, it will also accept the string `dog`. However, it will not accept any other strings, and so its language is simply the set of words $\{cat, dog\}$.

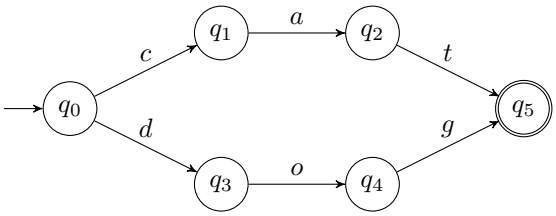

Figure 2: An example FA

Images as Finite Automata

We now digress to show how to describe black and white images using so-called quadtrees (see Fig. 3). Any 4×4 square is assumed to be numbered from 0 to 3, as in the given figure. The pattern repeats, so that a $4k \times 4k$ square adds another digit to these individual 'addresses'. Hence, the address of the black pixel in Fig. 4 has the value 3203. Each of the addresses of the black pixels in an image, can

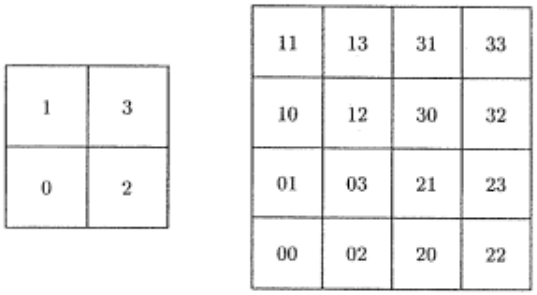

Figure 3: Pixel addressing scheme

now be considered as a string over the alphabet $\{0, 1, 2, 3\}$. If one therefore takes an image (assume that it is square with both width and height a power of two), then one can determine the address of each black pixel. Consider each such address as a string over the alphabet $\{0, 1, 2, 3\}$. Then one can find the corresponding FA that would recognize all these strings, but would not recognize any other strings.

Typically, for a highly repetitive image, the FA will require far less memory to store than the original image – therefore, the FA is a compressed version of the image. This compression corresponds to Phase

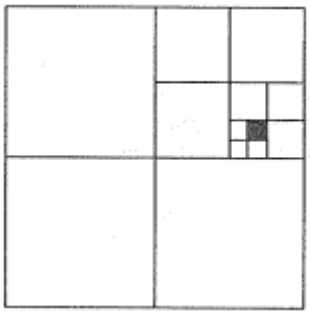

Figure 4: Address example: pixel with address 3203

2 of the project, and the algorithm can be found in the accompanying article.

Given an FA as described above, one can unpack all the strings that are accepted by the FA into pixels in an image – this is the decompression phase, corresponding to Phase 1 of the project. It is up to the student to decide how to produce the language (set of strings) of the input automaton, and use it to produce an image.

It is highly recommended that you take random small images, and work through the compression and decompression by hand, to make sure that you understand the principles.

Multi-resolution Images and Limitations

Automata produced by compressing repetitive images such as Figure 1 will contain cycles, or loops. An FA with one or more cycles, will have a language containing an infinite number of words or strings – since one can traverse the cycle an arbitrary number of times, and produce strings of any lengths. To decompress such FA, then, requires that you choose what lengths of strings to consider. The chosen length, will then also affect the resolution of the image – a word length of n will produce an image with resolution $2^n \times 2^n$.

It was also mentioned that highly repetitive images produce FA using far less memory than the original image. This is only true, however, if the algorithm is applied perfectly. Consider again Figure 1 – it is evident that the three sub-triangles are near identical to the full image, and thus could be described using the same language of set of states. However, to a computer program, the smaller triangles will have a lower resolution, and thus will be distinguished from the original image – the program often makes no use of cycles in its automaton at all. This leads to inefficient encoding of repetitive images, and can often cause the FA encoding of large images to take more memory than the original image. For the purpose of this project, however, the aim is simply to apply the algorithm accurately, and not to produce efficient compression software.

Phase 3 of the project will require you to account for cycles in FA, by enforcing a word length – passed as a command-line argument – on your decompression. Furthermore, to account for the loss of cycles mentioned above, some image processing will be done to add cycles to the compressed FA, to simulate compressing images to multi-resolution FA.

Git repository

Everyone will be assigned a git repository for the project. This is where you will do everything related to the project. An email should be sent to your student number once you have been granted access to your repository.

A handy guide to follow to clone and set up your repository once received: Git Beginners. Make sure you follow it in detail.

Once your git repository has been set up, ensure that you follow the given directory structure in the root of your repository (note: you will need to make the directories yourself for your local repository):

- `src:` the directory where all of your Java files will be contained. This includes `Compress.java` and **any other** files needed for your program to run;
- `bin:` the directory where the compiled Java `.class` files go. **These may not be pushed to the repository – if you do, you will be penalised**. This is just for marking purposes, and the script will ensure that this directory is present on the remote repository during marking;
- `out:` the directory where you will save any output produced by your program (save your output in text files instead of writing to standard output). Again, no output files may be pushed to the repository. This is just for marking purposes, and the script will ensure that this directory is present on the remote repository during marking;
- `tests:` the directory containing some of the input test files, their output files and the marking script itself called `testscript.py`. Do not push the tests directory to the remote repository, Git has access to a remote version of the script for pipeline and marking purposes.

Testing your own work

All of these directories must be present in your repository's main directory if you wish to run test cases yourself. Note that the `tests` directory is not provided directly with the git repository, but will be uploaded onto sunLearn. You will need to download this directory and put it in your repository as above. You must not upload this directory to the remote repository.

The marking script `testscript.py` will be provided inside `tests/`. This script is the same script that will be used for the final evaluation, but only a small subset of all test cases will be provided for each mode. This script can be run manually if you wish to check that your implementation will pass the provided test cases. This is in no way an accurate description of your final mark (since it is only a small subset of the test cases). However, it is valuable to confirm that your input and output formats are correct, and that you followed the specifications properly.

Within a file called `cases.py` you can see the list of test cases if you wish to run them manually instead of using the script, or if you wish to add custom cases. To see how to run the testing script, see the `README.md` file.

Output naming convention

Each mode will require saving your output to a text file. See the table below for the output names required for each mode. This is essential, as it is required by the marking script, so not following this means we cannot mark your submission.

All files must be saved in the `/out/` directory.
Text files must be saved as `.txt`, and images as `.png`.

Key	Mode	Name extension
1	Decompression	_dec
2	Compression	_cmp

Example

If you are given the input file `test1.txt`, then the output for the decompression must be `test1_dec.png`. For compression on input `test2.png`, the output must be `test2_cmp.txt`.
Phase 3, multi-resolution, will also use these two extensions for Decompression and Compression respectively. See the full output specifications for more details.

Style

You are expected to follow the prescribe style, which is a combination of Google Java style and Sun Code Conventions. Note, a line length of 100 is allowed, and TODO comments will be whitelisted for hand-ins

up to (and including) the midterm. Your project will be run through a style checker at hand-ins, which will contribute to part of the mark. This style checker also runs after every commit to your repository. You will also be required to document your program using the JavaDoc style and format.

GUI

There is no GUI necessary for the git handins, as the testscript runs on terminal input and output. However, you must implement a basic GUI for demo purposes. This GUI must simply be able to show the input and output next to each other. Any other features are optional. No extra marks will be awarded for fancy GUIs at all.

The first command-line argument will be used to distinguish between these two options. An argument of 0 indicates no GUI, and will be used for marking with the test script. Ensure that no GUI output is produced in this case. An argument of 1 indicates GUI, and will be used in the demos. It is up to you to decide and describe the command line arguments and output methods of your program in this case.

Phase 1: Decompression

Phase 1 involves reading in a text description of a finite automaton, and producing the corresponding image. The text description will follow a standard format, described below. You must read in the file, process the content, and design a data structure to contain your FA. You will then need to design an algorithm to process your automaton and produce the set of input strings that it recognises. Using this set, you can create a black and white image, using the words as the addresses of the black pixels.

Input

Mode 1 is run with

```
java -cp bin Compress 0 1 f <filepath/filename.txt>
```

The first argument indicates that no GUI is to be used, the second that this is mode 1, and the third indicates that no multi-resolution processing is necessary (See Phase 3).

The input file will contain an automaton description with the following format:

<number of states>
<list of accept states, space separated>
<List of transitions:
    <origin state> <destination state> <alphabet character>
    <origin state> <destination state> <alphabet character>
    ...
>

Output

An image file (`.png`) with the same name as the input file, but with _dec name extension, containing the black and white image produced by the FA.

For example, if we run `java -cp bin Compress 0 1 f test1.txt`, we expect the output file `test1_dec.png`.
`Input:`
5
4
0 1 0
0 1 1
0 2 2
0 3 3
1 4 1
1 4 2
2 4 1
3 4 0

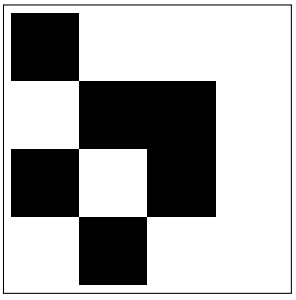

Figure 5: Output file

Phase 2: Compression

Phase 2 involves reading in an image, and applying the algorithm from the accompanying article, briefly summarised below as well. Again, you are advised to use some custom designed data structure to contain your FA, from which you can then write to the output file.

Pseudo-algorithm:

- Variable: n is number of states; i is next state; L_j is language of state j
- for each alphabet character a
  - if for some state j, it holds that $L_j = a^{-1}L_i$, then add a transition from i to j on a
    - Note: think of the languages in terms of sets of words. Then $a^{-1}L_i$ is a set of the words from L_i with character a removed from the start.
  - else create new state, add transition from i to n on a and increase n
- terminate when all states are processed. State 0 is the start state, and any state j with $\epsilon \in L_j$ is an accept state

Input

Mode 2 is run with

```
java -cp bin src.Compress 0 2 f <filepath/filename.png>
```

The input file will be a black and white image to be compressed.

Output

A text file (`.txt`) with the same name as the input file, but with _cmp name extension, containing the textual description of the FA, using the same format as the input files in Phase 1.

For example, if we run `java src.Compress 0 2 f test2.png`, we expect the output file `test2_cmp.txt`.

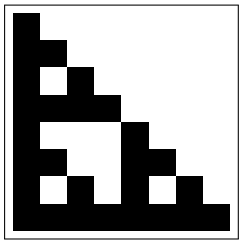

Figure 6: Input image

```
Output:
4
3
0 1 0
0 1 1
0 1 2
1 2 0
1 2 1
1 2 2
2 3 0
2 3 1
2 3 2
```

Phase 3: Multi-resolution

Phase 3 revisits the work done in phases 1 and 2, adding extra functionality, investigating various image processing techniques, cyclic automata, and the concept of Lossless vs Lossy compression.

For part A, or decompression, the approach is quite simple: account for input automata that contain cycles, by taking an additional argument specifying the word length to use. Take note to ensure that the output image has the appropriate resolution, related to word length.

Part B, compression, is slightly more involved. We aim to obtain an FA that can reproduce, or at least estimate, the input image at higher resolutions. To achieve this, we need to make an informed choice of where and how to include cycles to our automaton, or alternative ways to allow different

resolution outputs. Note that these approaches need to be as general as possible - i.e. they should take a fixed resolution image, and return an automaton that can produce *any* higher / lower resolution using Phase 3 decompression. We consider the following methods, specified to your program by command line argument:

1. Sierpinski triangle magnification: We can increase the resolution by subdividing each black pixel as usual, and making 3 of the 4 sub-quadrants black - i.e. each pixel gets replaced by 4 sub-pixels, increasing the dimensions by factors of 2. The white pixel corresponds to the lightest quadrant of the original image. This produces an effect similar to a Sierpinski's triangle pattern.

2. Checkerboard: We increase the resolution by placing copies of the original image, in each of the four quadrants of a new, higher resolution image. This produces recursive images like the checkerboard example in the accompanying article.

3. Reduce: another aim might be to reduce the resolution of an image. To do this, group 4 pixels together to form a (sub)quadrant of an image, and make this new pixel black iff at least one sub-pixel was black. Doing so once produces an image with dimensions decreased by a factor 2. Note that your approach should be general enough to produce images of arbitrary (lower) resolutions.

Your aim here is not to produce a different resolution image, but rather modify the FA produced by Phase 2. This modified automaton, when run through multi-resolution decompression, must then be able to produce different resolution images according to the specified method.

You will notice that for some examples, these multi-resolution approaches are able to correctly produce higher / lower resolution images, exactly as intended. This is referred to as Lossless compression - no information about the original image is lost through the compression - decompression process. However, in some cases some fine details of the image might be lost - this can be especially noticeable when attempting to decompress an image to its original resolution. This is called Lossy compression - some information gets lost in the process, resulting in lost, or added, black pixels. Your aim is not to always achieve Lossless compression, but rather to get your approaches as accurate as possible.

Note: The test script will test multi-resolution compression by running your program twice – first to produce the modified FA, secondly to test this automaton with decompression, with a specified resolution/word length. Thus, it is not so important *how* the FA is modified, but rather the end result that is produced. It is also important that part A of Phase 3 (decompression) works correctly.

Input

Multi-resolution functions with both modes 1 and 2, each with their own additional parameter.
Decompression is run with

```
java -cp bin src.Compress 0 1 t <word len> <filepath/filename.txt>
```

and compression is run with

```
java -cp bin src.Compress 0 2 t <method> <filepath/filename.png>
```

NOTE: The multi-resolution flag should not be case sensitive (i.e. your program should handle arguments `t` and `T` in the same way).

Output

Phase 3 uses the same output format and conventions for decompression and compression as Phases 1 and 2; that is, multi-resolution decompression produces a `.png` image, with the _dec name extension, and similarly for compression with `.txt` files.

Error handling

Some error handling will be required. That is, a few test cases will have invalid input parameters or input files and an error message must be sent to standard **error** (not standard output). The errors below are listed from highest priority to lowest priority – that is, if the mode is a string and not an integer, then `invalid argument type` must be displayed, and not `invalid mode`.
Here is the list of errors that must be catered for, as well as their expected outputs:

INPUT ERRORS

1. **Invalid number of arguments**: If the incorrect number of arguments is given, then the following error must be displayed:

   ```
   Input Error - Invalid number of arguments
   ```

2. **Invalid argument type**: If an argument is of the wrong TYPE (for example, a mode of "abc" is given), then this error must be displayed:

   ```
   Input Error - Invalid argument type
   ```

3. **Invalid GUI**: If the GUI is not 0 or 1, then this error must be displayed:

   ```
   Input Error - Invalid GUI argument
   ```

4. **Invalid mode**: If the mode is not 1 or 2, then this error must be displayed:

   ```
   Input Error - Invalid mode
   ```

5. **Invalid multi-resolution flag**: If the multi-resolution flag is not in $\{f, t, F, T\}$, then this error must be displayed:

   ```
   Input Error - Invalid multi-resolution flag
   ```

6. **Invalid file**: If the file name is given, but the file cannot be opened or does not exist, the following error must be displayed:

   ```
   Input Error - Invalid or missing file
   ```

DECOMPRESSION ERRORS

7. **Invalid word length**: If the multi-resolution word length is negative, the following error must be displayed:

   ```
   Decompress Error - Invalid word length
   ```

8. **Invalid file formatting**: If the file contains invalid automaton formatting - strings instead of integers, or incorrect number of elements per line - the following error must be displayed:

   ```
   Decompress Error - Invalid automaton formatting
   ```

9. **Invalid accept state**: If an invalid accept state (one not corresponding to a valid state number) is given, the following error must be displayed:

   ```
   Decompress Error - Invalid accept state
   ```

10. **Invalid transition**: If a transition includes an invalid state number, or an invalid alphabet character, the following error must be displayed:

    ```
    Decompress Error - Invalid transition
    ```

COMPRESSION ERRORS

11. **Invalid image**: If the image given is not square or contains pixel values other than black or white, the following error must be displayed:

    ```
    Compress Error - Invalid input image
    ```

12. **Invalid multi-resolution method**: If the multi-resolution method given is not in $\{1, 2, 3\}$, the following error must be displayed:

    ```
    Compress Error - Invalid multi-resolution method
    ```