University of Rennes 1

Course in Operating Systems 2022-23

Course Project

Year 2022-23

Binary Graph Encoding

Date: **24/11/2022** - Project: **6**

Track: **CNI**

Student: **Jacobo Javier Galache Gómez-Coalla**

Index

Introduction

This document is a report for the course project of the subject in Operating Systems: Project 6. This project requires the implementation of a software program that will be able to take a properly structured graph object, encode it into a simple sequence of bits and be able to retrieve the original graph from the encoded sequence. It has to be implemented in java and has to support multithreading with an emphasis on performance optimization. In this report, there is a detailed description of the main software components implemented, a bird's eye view of how the entire system operates as well as a comparison between the sequential and multithreaded performance of the implementations.

System Overview

This section consists of a description of how the implemented system works, including some illustrations and examples to better understand the implementation.

To start with we have to quickly overview how the graphs are modeled in order to understand how to process them back and forth from binary. The graphs are implemented as a separate class inside of the graph package. In this class we can find a total of 3 parameters:

- directed: it describes whether the graph is or not directed
- vertixMap: it stores a set of key value pairs where each key is the index of each vertex and the value is the list of vertices with which it has an edge
- nVertix: it stores the total number of vertex the graph has

```
private boolean directed;

private Map<Integer, LinkedList<Integer>> vertixMap;

private int nVertix;
```

When the graph is initialized it requires a boolean value as a parameter which will set the directed variable. The class implements a standard suit of methods usually associated with graphs, it also contains some extra functions that will prove useful in the implementation of the project:

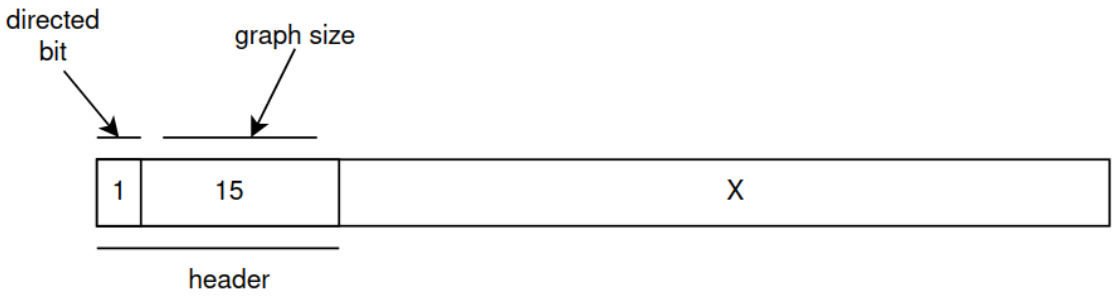
- **toString**: it allows for the conversion of the graph into a string, enables easy visualization of the data for testing purposes
- **setDirected**: it allows for toggling the variable directed from the graph
- **getVertixList**: it returns the list associated with a vertex
- **getSize**: it returns the size of the graph

The binary encoder and decoder functions are implemented in a class called Binary, inside of the package binary. Inside the most obvious functions to take notice of are getBinary and getGraph, which implement a big part of the logic that goes into the functionality of the program. The project comes with a second class in the binary package. called ThreadedBinary. This class implements the multithreading into the sequential implementation of the program. In the following sections the components of the program will

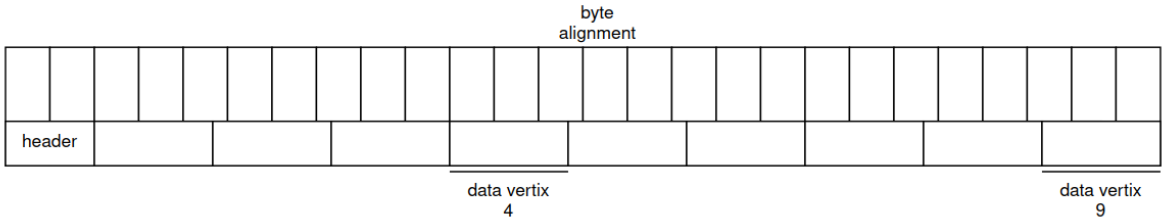
be broken down, but the following is an explanation of how the process is structured and how the encoding and decoding work fundamentally.

Encoding

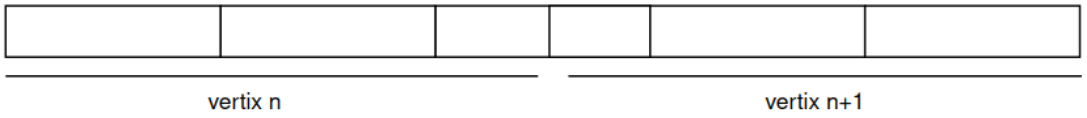
The encoding starts by checking the status of the graph. Some data is extracted such as the size of it or if it is directed or not. This information is encoded directly into binary into the header section of the binary array. In particular the header is chosen at a size of 16, utilizing the first bit for the directed bit and the other 15 to encode the size of the graph, which is the only information that will be stored in the header and the only information that will be stored in the binary encoding apart from the edge data.



It is the only information stored in the encoding because it is the only one we need to define the graph when decoding.
After the header has been set, we enter a loop that will run as many times as the number of vertices in the graph. It will create an auxiliary binary array that will be used to encode the edges of the graph. The edges will be encoded by a single bit each, where a value of 1 will signify that the edge exists and a 0 will mean that it does not. As such each vertex will have a section inside of the binary array that will define the edges it has, this section will be composed of as many bits as vertices the graph has. This way the total bit length of any one binary array will be the sum of the header length plus the graph size squared. Once the encoding has finished, it will be shifted the corresponding amount to fit the placement set by the previous sections of the encoding.



In the illustration we can see the structure of the binary array. As we can see the alignment with the bytes is not guaranteed when setting the binary encoding, as such adjustments will be necessary to fit the data to the array.



As an example, in the figure above we can see how the vertex n ends in the middle of a byte, which will require the vertex n+1 to be shifted to the left to fill the byte before using the following ones.

Once the data has been encoded and shifted accordingly, it will be added to the binary array by performing an OR operation between them in the relevant bytes. With this we finish the encoding and retrieve an array of bits that contains all the information from the graph.

Decoding

Decoding, similar to encoding, starts by treating the header, in this case we retrieve it from the binary array in its entirety. After it has been separated from the main array, there is a check of the directed bit, whose solution will be stored in a binary variable for later. The bit is set to 0 regardless of the original value and then the rest of the header is decoded into an int, this will define the size of the graph.

After obtaining this, the graph is initialized and we enter a loop. This loop will run from zero to the size of the graph. In it, there will be an auxiliary array where the relevant chunks of the vertices will be loaded to be decoded. Once the information has been added to the graph object, we will have the graph completely decoded.

Software Components

In this section there will be a description of the main modules that compose the main functionality of the program, mainly encode and decode.

getBinary(graph)

This function is the main function utilized to transform a graph into a binary array. This function has one parameter, it being a graph object. At the start of the function there are 2 variables being set, one is the size of the header, hardcoded to be 16, the other being the size of the graph, obtained from the graph object passed as a parameter. A small calculation will take place to ensure that the size of the graph is not big enough that the assigned header size is not enough to encode it in binary (that the graph size is smaller than 2^{15}).
Afterwards, the binary array will be created, and the header will be loaded into it. To do so we require the use of a function implemented called getBinary, that takes integers as parameters and returns the binary encoding for it. The header will be loaded offsetting the bytes so as to load the significant digits into the header (integers can take up to 32 bits to encode, we only take the 15 least significant digits, so we need to load only the last 2 bytes of the binary representation).
Following this, the directed bit will be set and after that, the entire rest of the array will be initialized with zeros. The next part consists of the creation of the auxiliary array that will be used to prepare the encoding for the complete binary array, this auxiliar array has to have a particular size because it has to account for the offset being applied, as such it will need to be the length of the ceiling function on the size of the graph over 8 plus one. This is because the size is declared in number of bytes, as such we have to translate the size of the graph into bytes and add one because it can be shifted over backwards or forwards depending on

the offset a maximum of 7 positions (if it shifts more than 7 positions it moves to another byte so it gets annulled).
Once inside the loop the offset is calculated as the size of the graph times the number of iterations done inside the loop. This is because each time a vertex has been processed, a number of x bits has been added to the binary array which correspond with the size of the graph (each vertex has X number of bits that define each of its edges). Then the function getByteFormatted will be called. This function returns a binary array with all the information ready to be inserted into the binary array.
After, there is a loop which goes through each and every byte from the auxiliary array and performs the OR operation with the corresponding byte in the binary array. The byte indexes are calculated by adding the looping variable of the nested loop with the offset over 8 (to transform it into byte index) and the header size over 8 (also to transform it into a byte index).
When these operations are done, the function returns the binary array.

getGraph

This function is the main function utilized to decode the binary array back into a graph object. The only parameter it has is a Byte array. The function starts by defining the header size, similarly to the getBinary, to be 16 bits, another variable passes this size into a byte index (dividing by 8). A new binary array is defined with a size of 4 (standard integer size), the header is loaded into it while minding the position of the bits inside the array. It is important to load the information aligned to the right (with the end of the header at the end of the Byte array) so as to be able to convert it successfully back into the integer that was encoded. There is a check for the directed bit, whose information is stored in a boolean variable to be used at the end of the function. The bit is set to 0 and the header is retrieved back into an int.
After this the graph object is declared to be directed. This is because with the current encoding, the edges will be encoded twice, once in the direction (x -> y) and another in the direction (y -> x) since every vertex has a bit reserved for every other vertex. Setting it to directed will allow for the adding of edges correctly, afterwards being possible to toggle the directed variable if necessary (if we set the graph to be not directed, the current implementation adds each edge twice since it was encoded twice as mentioned before).
Following this step, a new auxiliary binary array is defined with the same size as explained in the previous section of the document. A loop is entered that will run once per vertex of the graph. The offset will be calculated in a similar manner as in the previous section and another loop will be entered this time. The loop will run once per byte in the auxiliary array, loading the relevant bytes into the array to be processed. The index is also the same as explained in the previous section.
After loading the info, a function setEdgeInformation is called, that will add the appropriate edges to the graph. After the loops are finished, the directed status of the graph will be set to the correct value and the graph will be returned.

getBinary(int)

This function is used to encode integers into binary arrays compatible with the structure used for them. It returns a 4 byte array with the binary encoding.

getByteFormatted

This function is utilized to encode the data of a single vertex into a byte array. It receives a total of 3 parameters, the array length for the auxiliary binary array, the list of edges and the offset. It starts by creating a byte array where the information will be loaded, then it defines an array of constants, these constands, expressed in hexadecimal, define the 8 values that have only one bit with a value of one in the entire byte. This is useful to set individual bytes to 1 in the array.

```
Byte[] arrOfValues = new Byte[8];
arrOfValues[0] = (byte) 0x80;
arrOfValues[1] = (byte) 0x40;
arrOfValues[2] = (byte) 0x20;
arrOfValues[3] = (byte) 0x10;
arrOfValues[4] = (byte) 0x08;
arrOfValues[5] = (byte) 0x04;
arrOfValues[6] = (byte) 0x02;
arrOfValues[7] = (byte) 0x01;
```

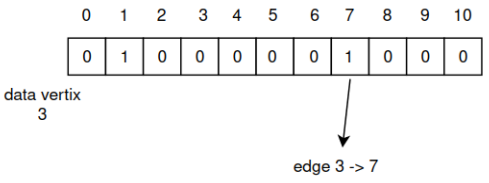
the auxiliary array is initialized to zeros and a loop is launched that will run through every item of the list of vertices, for each of the iterations, there will be a bit set on, this bit will be in a specific byte and with a specific shift. The byte is calculated by adding the integer of the list to the offset over 8 and the shift will be calculated by the integer of the list plus the offset and all of that modulo 8. The bits are set by calculating an OR gate between the auxiliary array's byte and the array of constants.

```
for(int v : data) {
        vertixData[(int)((v+offset)/8)] = (byte) (vertixData[(int)((v+offset)/8)] | arrOfValues[(v+offset)%8]);
}
```

After this, the auxiliary array is returned.

setEdgeInformation

This function serves to decode the information of a vertex, adding the corresponding edges to the graph. It receives 4 parameters. The byte array with the encoded information, the graph to be edited, the size of the graph and the iteration of the outer loop.
It defines the same array of constants as before, calculates the offset by multiplying the length and the iterations and calculating the modulo 8 over it. Then it enters a loop, this loop will run once per vertex for all the vertices of the graph (a total of length times). It calculates an AND operation between the data byte and the corresponding value of the constant array, if the result is different from 0 then it will set an edge between the edge set by the iteration and the value of the loop variable. This works because the outer loop represents which block of encoding is being processed (which vertex's edges we are decoding) and the loop variable represents the index of the bit in the block, which represents the edge with the same index.



After the loop ends, the method finishes.

binaryToString

This function serves to print binary arrays in a way that shows all the digits be it zeros or ones. It takes a single byte array as a parameter and will return a string that represents the value in binary of the array of bytes. It works by converting byte by byte the entire array, performing the AND operation with the value 0x000000FF beforehand. After it obtains the string from the operation it adds as many zeros as necessary to the right so as to make the length of the string composing each byte 8. It adds the string obtained to a general string and then returns it when finished.

Multithreading

The project put an emphasis on implementing multithreading as one of its objectives. As such, in this section the multithreading implementation will be discussed.

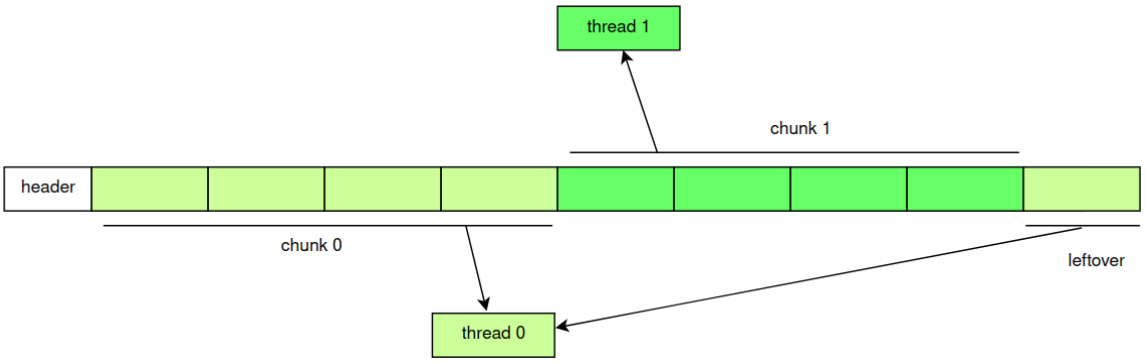
As mentioned before, multithreading is implemented in a class named ThreadedBinary. The class is merely composed of a replica of the functions mentioned in the previous section, however some of them have suffered some modifications to accommodate the implementation and execution of multiple threads.

The two functions which were modified were getBinary(graph) and getGraph. These functions now include a new section before the loops processing the edge information. This means that the header processing is still done sequentially and nothing has changed about that. After finishing the setup of the header and calculating the array length of the auxiliary arrays, there is a definition of a new variable, threadCount, that will be used in the initialization of a Thread object array called threadPool. After this initialization, a new loop will execute, each iteration will declare a new thread with a run function calling a new function extracted. After the definition, the name of the thread will be set to their index in the threadPool array and the thread will be started.
After the loop has finished and all the threads are running, a new loop will run, executing thread.join on each of the threads to synchronize them before returning the solution of the function.

Both functions feature the function extracted; however, they have different parameters, allowing for both to have the same name.

The functions extracted are modified versions of a refactored function extracted from the sequential version of the code. In particular, it takes the loops that went through from 0 to the size of the graph. The main difference is that they swap the start and end points of the vertex loops, running instead of all the way through, only the chunks assigned based on the total number of threads and computing chunks available. Apart from doing this division, there is a new loop which enables the assignment of the leftover sections of the computation to different threads. This refers to the parts of the array that are not assigned to any thread due to having a situation where the size of the graph is not divisible by the number of threads. In

this situation, I assign those leftovers to one of the threads which will process after it finishes its original chunk.



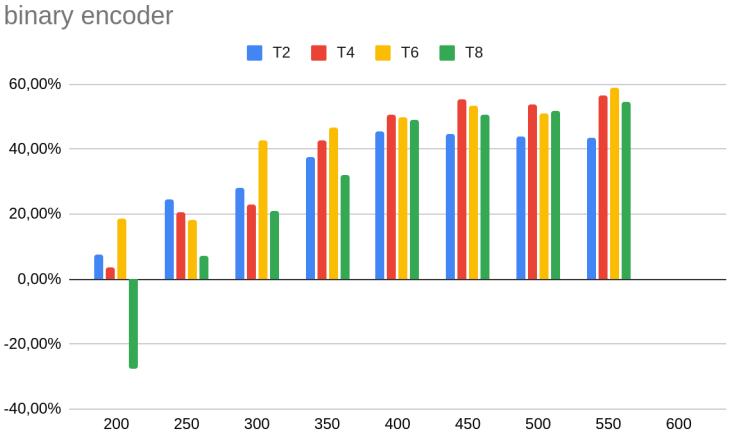
In my implementation, thanks to the usage of or gates to aggregate the results and separate lists per vertex, I can allow the race conditions to affect the execution of the code, since they will not negatively impact the result. As such, the graph object in the case of the getGraph function or the binary array in the case of the getBinary function will be shared objects being operated by multiple threads at the same time.

Results

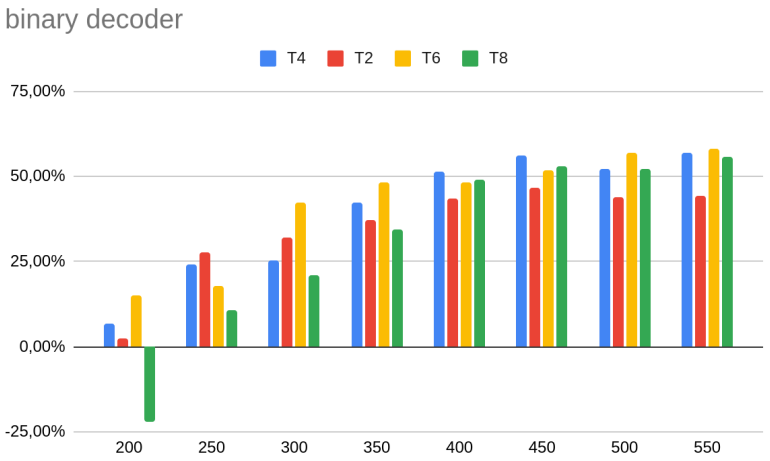
In this sections, a discussion of the results will be included, as part of the multithreading implementation.

Multithreading is utilized commonly to boost performance of programs and applications. In the following paragraphs the data extracted from executing tests on the implemented program will be discussed, focusing on the difference in performance between the sequential and threaded implementations.

First to note that all the tests have been performed in the worst case possible for a graph. With the current implementation of the program, the encoding of the graph will only require looping through existing elements of the lists of edges, meaning the less number of edges, the faster it will compute, this is why these tests have been done on graphs that have edges from all the vertices to all the vertices, meaning that for a graph of n vertices, it will have n^2 edges. Also to note, all the tests have been performed on a laptop with an i7 10 series with 4 cores, 8 threads.



We can observe in this first graph how the impact of more threads impact the performance of the execution to the runtimes. The graph shows the average performance boost obtained from running the threaded implementation compared to the sequential one. each of the tests was performed a hundred times and the average of the performance increase was saved. The performance increase is calculated by subtracting the threaded runtime to the sequential runtime, dividing the result by the sequential runtime and subsequently multiplying the result by 100.
It is interesting to observe how when the number of vertices is low, the performance of a big number of threads is worse than a lower number of them and also worse than the sequential implementation. It is important to also note there were significant fluctuations in the performance and the average is only a relatively significant representation of the real performance. As we can see however, at a higher number of vertices we can appreciate how multithreading really pulls its weight and manages to halve the computing speed.



As we can observe in the second graph, the same applies to the decoder, the performance takes a hit when utilizing 8 threads at such a low vertex count but increases as we increase it.

As a conclusion to this section, it is interesting to note how the best overall thread count for all the scenarios tested is 6 threads. I do not know exactly why that is but I theorize it might have to do with the split in usage of the CPU threads. Having 8 threads in total, 2 might be occupied with the operating system or other basal computation, and only 6 threads would be available to compute with (talking about intel's Hyperthreading).

Conclusions

In this section the conclusions and personal impressions obtained from developing and finishing the project are included.

First I would like to say I did have a good time developing the project. I found it interesting when I first chose it and found it entertaining to think through the different issues that came up during development. I also did like the management style with the project, allowing the students to work on their own to solve the problems, only contacting the teacher in case a

doubt or problem arose really did give a good sense of agency and freedom when modeling, designing and ultimately developing the project.
I would also like to express my regret at not being able to finish completely all of the objectives I set out to accomplish, mainly due to time constraints, referring mainly to the implementation of the weighted graph decoding and multithreading.
Overall I did find the project to be worthwhile, from the start when we were given the freedom to choose between the different suggested projects, to the entire way of handling the TPs to work on the project. I also learned a lot about how the management of binary encoding works. In general a great course assignment.