Документация на проект за проект по СДП-лекции - Алгоритъм на Хъфман

Проекта е имплементация на Алгоритъма на Хъфман за компресия на низове без загуба на данни.Освен тази документация в кода на самия поект има подробни коментари, които могат да бъдат извлечени в HTML от приставката за code::blocks – doxygen.

Целта на проекта е да се прочете низ от текстов файл, да се компресира , да се запише полученото компресирано съобщение , както и построеното хъфманово дърво, така че след край на изпълнение на програмата всичко нужно за по-късна декомпресия на кода да е налично.Също така се иска проекта да се способен да прочете от файловете този код и самото дърво и да възстанови началния низ.

Двете главни фунцкии имплементиращи тези процеси са:

void Compress(const char* from,const char* code_to,const char* tree_to,const char* bool_code_to,const char* int_code_to) – тази функция чете низ от файл с име from,записва полученото компресирано булево съобщение (корирания низ) в двоичен файл с име code_to, а полученото дърво в процеса се записва в двоичен файл с име tree_to ,ако специално са указани последните два параметъра (те по подразбиране са NULL) в текстови файлове с съответните имена се записват получените булево компресирано съобщение(компресирания низ) и целочисленото компресирано съобщение (булевото компресирано съобщение преобразувно в десетична система (секи 8 бита=1 десетично число)). В процеса на изпълнение на функцията се случва следното – създава се обект от клас Tree от конструктора-tree_constructor_three(…),който от своя страна първо чете ред по ред низа от текстовия файл, събира целия низ в буфер, от този буфер строи честотна таблица (обект от тип Table) .След като таблицата е построена се създава масив със сичките и елементи които представляват и листата на дървото, после се строи самото дърво от този масив по следния начин – взимат се двете листа с най-малко повторения и се създава нов обект от тип Vertex чийто ляв наследник е листото с по-малко повторения,а десен- това с повече,ь. След това тези две листа се маркират като обходени и този нов обект от тип Vertex бива добавен в масива и процеса се повтаря.Това продължава докато не остане само един немаркиран обект от тип Vertex , който реално е корена на дървото.След като дъвото е построено, последователно в обект от тип Stack се записва пътя (в обратен ред) от корена на дървото до листо на дървото.След това като имаме пътя определяме кода на това листо и го записваме на съответния обект от тип Vertex в таблицата на четота.След като имаме таблицата и дървото компресираме входния низ като за всяка сещната буква от низа ползваме съответния и код от съответния Vertex обект от честотната таблица с същата буква.След като булевия код е готов образуваме и целочисленип код като за всеки 8 бита от кода записваме целочисленото число което се получава от тези битове.Също така тъй като за последното число може да останат по малко от 8 була , при евентуално преобразование от целочислен към булев код няма да се знае колко начални нули ще трябва да се сложат, затова в края на кода се записва едно допълнително число,което е бройката на булове от което се е получило последното число от кода, така при евентуално преобразуване от целочислен отново към булев код ще се знае колко начални нули да се сложат.След като таблицата,дървото и кодовете са готови, всеки осем бита от булевия код се записват в обект от тип PackedByte ,така че 8 бита да се запишат в 1 единствен байт вместо в 8 отделни,след което дължината на отправния низ,дължината на компресираното булево

съобщение (компресирания булев низ) и всички получени PackedByte-ове се записват в двоичен файл.Накрая тъй като обектите от клас Tree съдържат указатели като член данни, той не може директно да се запише в файл.Затова се създава обект от клас NodeTree, който е еквивалентен на хъфмановото дърво от обекта от клас Tree, само че не използва указатели за да запази релацията между върховете в дървото, и може да буде сериализирано(записано в файл директно).С това процеса на компресия приключва.

Другата главна функция е: void Decompress(const char* TreeData,const char* CodeData,const char* DecompressTo). Тук TreeData е името на файл от който да се прочете обекта NodeTree записан в двоичен файл при компресията на низа, който предстои да декомпресираме.CodeData е име на двоичен файл в който е бил записан компресирания булев код ,под формата на PackedByte обекти, който предстои да декомпресираме.DecompressTo е името на текстов файл, в който да бъде декомпресирано съобщението.Процеса на декомпресия е следния – четат се обектите PackedByte от двоичния файл, възстановява се от тях булевия код , създава се и обект от клас NodeTree ,след това се върви паралелно по булевия код и по дървото, като когато се стигне до листо се записва буквата държава в това листо и отново се тръгва от корена на дървото.Накрая получения декомпресиран низ се записва в текстов файл.

Ето кратко описание на функциите и целта на класовете ползвани в проекта,за по подробни спецификации на функциите на тези класове виш документацията на кода (извлечена с приставката за code::blocks doxygen) и самите коментари в кода.

Vertex:

Objects of the Vertex class are used to build the Huffmann Tree in the Tree class. A Vertex contains data for the Vertex itself, pointer to left successor and pointer to right successor in constrast to the objects of the Node class which contain no pointers and therefore can be serialized easily.Objects of this class are used to build the frequency table, the Huffmann tree and the serializable Nodes used for the serialiszble NodeTree.This class is in the basis of the structure of the program.

Table:

This class represents the frequency table used to build the Huffmann tree.Its purpose is to count how many times each letter is found in the input string, create a Vertex for each such letter, and store them in a Vertex array.It is inheited by the Tree class for later use.

Tree:

This class builds the Huffmann Tree for the compression. It inherits a Table containing the leaves of the tree the Tree is built from the bottom-up ,after that codes are assigned to all leaves in the table, then the string is compressed using the codes .If specified to do so by the constructor used, the Tree is transformed in a form appropriate for serialization and is written down in a binary file. The compressed string into code is also written down in a binary file. That way after the program is closed, the string can still be restored later on when needed.

Box (struct not a class):

Elements held in the Stack class. Holds a Vertex and a pointer to the previous Box.

Stack:

Stack data structure, with standard implementation used to store Box objects

Node:

Objects of this class are similar to the objects of the Vertex class. Nodes are constructed from Vertices, keeping their letter, and relations. The main difference is that Nodes do not need pointers to store the relations.ID-s (Key values) are used to store the relations between Nodes. This way objects from Node class can be serialized and deserialized , while objects of the Vertex class cannot.

NodeTree:

Object of this class represent the HufmnnTree. Object from this class are generated from objects of Tree class, keeping the data of the Vertices in the Tree and their relations, by constructing Nodes from the Vertices .The difference between Tree and NodeTree is that Tree contains pointers in its private members in order to store the relations between the Vertices and cannot be written down in a file, while Objects from NodeTree class use ID-s (keyvalues) to store these relations and thus can be serialized and desirialized to/from files.Also objects from this NodeTree class are used for Decompressing already compressed with Tree ojbects strings.

PackedByte:

Since we cannot directly access bits .We cannot write down Bools in a single bit, instead the Bool is being written in a Byte, and the Byte is being written down. To avoid this, and write down Bools in such a way that each bool would only take up 1 bit of memory-space instead of a whole byte, a special structure is needed. Objects of the PackedByte class are merely unsigned chars(1 byte=8bits).For the class we have operation which can access the nth bit, change the n-th bit to 1 , or change the n-th bit to 0.That way we can store 8 bools in a single Byte, write it down ,and later when we read it we can access each of those bools in the form of bits of the Byte. There is a constructor for the class which receives an integer 0-255 and transforms it into a byte (unsigned char) by representing that integer as a number in base 2.

Някои от нетривиалните проблеми срещнати повреме на разработка на проекта са следните.

1)Как точно да се запише компресирания код така че да вскеи bool да заема 1 бит вместо 1 байт(тъй като не може да достъпваме директно битове).Проблема е решен като се използват обекти от клас PackedByte които могат да ползват побитови операции.Обект от тип PackedByte представлява просто един unsigned char на който може да достъпваме и променяме n-тия бит ползвайки маски и побитови операции.Така записваме 8 (или по-малко) bool-ове в един PackedByte и записваме само този 1 PackedByte с размер 1 байт вместо 8 отделни байта за всеки един от bool-овете.

2)Друг нетривиален проблем е сериализацията/десериализацията на дървото на хъфман.Не може директно да запишем обект от клас Tree защото в него релацията между върховете е

изразена с указатели (за по-лесна работа с класа).Тъй като не може да записваме указатели в класа. За сериализация и десериализация се ползват обекти от клас NodeTree констроирани от съответни обекти Tree. Обектите от клас NodeTree не боравят с указатели за запазване на релацията между върховете в дървото а ползват ID (key-values) тоест всеки Node разполага с масив от 2 key-value за ляв и десен наследник.Така обектите от клас NodeTree не са толко функционални колкото тези от клас Tree, но са достатъчни за сериализация и десериализация без загуба на данни и релация между върховете.

Някои предложения за бъдещо подобрение на проекта :

Възможно е с проекта да се компресират не само низове но и други файлове-примерно може да вземем един mp3/jpeg файл, да го прочетем байт по байт, да преобразуваме всеки един от тези битове в число (0-255) да преобразуваме това число в char (extended ASCII table).След това да запишем получения странен низ в текстов файл.Така по вече имплементираните фунцкии може да компресираме този низ.След това чрез вече направените функции можем да възстановим този низ.След това трябва отново всеки char да обърнем до число от 0-255 (ASCII extended) кода на символа, да построим PackedByte от това число и да запишем тези PackedByte-ове в файл чието разширение да съвпада с това на начално подадения файл.

Освен това може да се постигне още по добра компресия.Тъй като понякога е възможно за определена буква която се среща сравнително рядко(примерно само 1000 пъти в дадена книга) кода на тази буква може да е по-дълъг от 8 бита тоест ще се получи, че кодираме един char(8 бита) с повече от 8 бита,което не намалява а увеличава размера.Ако в случаите, когато кодът на дадена буква е по-дълъг от 8 може да помним самата буква вместо кодът.Така обаче ще се получи нехомогенност в компресираното съобщение – то ще се състои и от булове и от символи и така ще се усложнят процесите на компресиране и декомпресиране на информацията.