Gamescrafters Documentation
Level Files

Deepa Mahajan

Version History

2006.11.10 - Version 1.0 - First version created.

Contents

1 Purpose of Level Files

Level Files are files which store all reachable positions for a given tier. They are meant to be used in conjunction with the Tier Gamesman API and are also used in the ODeepa Blue Project. These files are used by the Tier Gamesman Solver because it provides the solver with a list of all positions that are accessible.

2 Definitions

Below are key definitions to any specific Level File terminology that will be employed.

2.1 Full Level Files

Level Files contain a header and then data regarding which POSITION's are valid in a given tier. The way the data is stored depends on the type of level file. The full level file is named `/data/mgame_opt_tier.dat.gz`

2.2 Partial Level Files

Partial Level Files are level files that contain only a limited portion of the full tier's level file. These files are typically created during use with ODeepa Blue when each computer is generating a partial level file based upon the positions they are responsible for generating. The partial level file is named `/data/mgame_opt_tier_startVal_type.dat.gz`

2.3 Intermediate Level Files

Intermediate level files are generated when the user calls writeLevelFile because all four types of level files are generated. These files are identical to the full level file, but have a different naming convention just so they can coexist while determining which one to label the full level file for that tier. They are named `/data/mgame_opt_tier_type.dat.gz`

3 Structure of the Four Defined Level Files

3.1 Purpose

When considering large games such as Bagh Chal or Quarto the very size of each tier and its level file can be a limiting factor as to whether the game is solvable. Although level files utilize gzip compression, the following four files formats for level files was created in order to exploit our knowledge of the data being stored. The aim in reducing the file size is to allow for files that can be stored on a computer so that Tier Gamesman can load the files for solving purposes or to allow for files to be transferred between computers for any use of ODeepa Blue Parallelization.

3.2 General Structure

Any lines that begin with a # are comments. The first line contains a start check bit. This occurs before any comments. After this line, commnets may appear anywhere. The second and third lines contains any header information. The second line contains only the level file type (0, 1, 2, 3) and a new line character. The third line includes the minHashValue and maxHashValue. The minHashValue value is the minimum value of all the values in the tier. The maxHashValue is the

largest/maximum number value of all the values stored in the file. In types 1, 2, and 3 all values that are stored are stored where there is a global offset of the minimum legal POSITION. This means that for each position in the file (#1, #2, #3,) the file contains the values (0, #2-#1, #3-#1, .). Lastly bitsPerPosition is the value of the number of bits it takes to store the largest number value - the offset. All values printed in the file are normalized with 0's appended to the front of the values in order to keep them the same bit length. The header is separated from the data by the single character 'L̂' = char code 0x12. The files also only contain values from between minHashValue and maxHashValue. After the data, another 'L̂' = char code 0x12 is placed and then the final check bit '1'.

3.3 Type 0

A Type 0 level file essentially contains all of the positions that are reachable in that Tier. It has very minimal optimizations and contains the values in ASCII in order to enable easy reading and testing of level file generation. An example of a general file follows.

mGameName_Opt#_Tier#_Type0.dat.gz

```
1
### GAMESMAN Bagh Chal Type 0 Level Files by Deepa Mahajan
### Doc on level file formats, see doc/LevelFiles.txt
### 2006-08-02 @ 09:00 PST
0
minHashValue maxHashValue 0x12
Ascii positions separated by spaces 0x12
1
```

3.4 Type 1

A Type 1 level file essentially contains all of the positions that are reachable in that Tier. It has very minimal optimizations. An example of a general file follows. The header is stored in ASCII but the positions are in binary. Since there is a set bitsPerPosition, any position with fewer bits than that has 0's appended to the start to normalize the position nolength.

mGameName_Opt#_Tier#_Type1.dat.gz

```
1
### GAMESMAN Bagh Chal Type 1 Level Files by Deepa Mahajan
### Doc on level file formats, see doc/LevelFiles.txt
### 2006-08-02 @ 09:00 PST
1
minHashValue maxHashValue 0x12
Positions in binary not separated by spaces 0x12
1
```

3.5 Type 2

A Type 2 level file essentially contains all of the positions that are NOT reachable in that Tier. An example of a file follows. The header is stored in ASCII but the positions are in binary. Since there is a set bitsPerPosition, any position with fewer bits than that has 0's appended to the start to normalize the position nolength.

```
mGameName_Opt#_Tier#_Type2.dat.gz
     1
     ### GAMESMAN Bagh Chal Type 2 Level Files by Deepa Mahajan
     ### Doc on level file formats, see doc/LevelFiles.txt
     ### 2006-08-02 @ 09:00 PST
     2
     minHashValue maxHashValue 0x12
     Positions in binary not separated by spaces 0x12
     1
```

3.6 Type 3

A Type 3 level file essentially contains a list of all the positions from the start to the end of the level. However, they are stored such that a reachable position is a '1' and an unreachable position is a '0'. An example of a general file follows. The header is stored in ASCII but the positions are in binary.

```
mGameName_Opt#_Tier#_Type3.dat.gz
     1
     ### GAMESMAN Bagh Chal Type 3 Level Files by Deepa Mahajan
     ### Doc on level file formats, see doc/LevelFiles.txt
     ### 2006-08-02 @ 09:00 PST
     3
     minHashValue maxHashValue 0x12
     1's and 0's in a row representing reachable and non-reachable positions 0x12
     1
```

4 API Functions and Variables

4.1 Functions used to Read/Write Level Files

4.1.1 WriteLevelFile

Description
`WriteLevelFile` reads an array of hash values and returns a level file. This file may be of any format type. The function optimizes for size by generating all types sequentially and comparing size of file after each file is generated. If the last value in the file is not a solitary '1' on the last line (preceeded by a 0x12) then the file is corrupted. The array of input hash values is in the form of a BITARRAY. A BITARRAY basically has each bit be a 1 if the position is reachable and a 0 if the position is not reachable.
Arguments

`char* compressed_filename`
`BITARRAY *array`
`POSITION startIndex`
`POSITION endIndex`
Return Values
`int success`: 0 for success and 1 for error

4.1.2 ReadLevelFile

Description
`ReadLevelFile` takes in a level file and returns a BITARRAY of available values. The array starts from the level files minHashValue and continues until its maxHashValue. These values must be retrieved by the calling function by using the getLevelFileMinHashValue and getLevelFileMaxHashValue functions. The level file may be of any type from 0 to 3. The BITARRAY array's length must be determined by the parent function by giving it a length of (maxHashValue-minHashValue)/8 bytes.
Arguments
`char* compressed_filename`
`BITARRAY *array`
`int length`
Return Values
`int success`: 0 for success and 1 for error

4.1.3 getLevelFileType

Description
`getLevelFileType` takes in a level file and returns the type it is.
Arguments
`char* compressed_filename`
Return Values
`int type`: 0 , 1, 2, or 3

4.1.4 getLevelFileMinHashValue

Description
`getLevelFileMinHashValue` takes in a level file and returns its minHashValue
Arguments
`char* compressed_filename`
Return Values
`UINT64 minHashValue`

4.1.5 getLevelFileMaxHashValue

Description
`getLevelFileMaxHashValue` takes in a level file and returns its maxHashValue

Arguments
`char* compressed_filename`
Return Values
`UINT64 maxHashValue`

4.1.6 getLevelFileBitsPerPosition

Description
`getLevelFileBitsPerPosition` takes in a level file and returns the number of bits per position used to store the positions.
Arguments
`char* compressed_filename`
Return Values
`UINT64 bitsPerPosition`

4.1.7 isValidLevelFile

Description
`getLevelFileBitsPerPosition` takes in a level file and whether it contains both check bits
Arguments
`char* compressed_filename`
Return Values
`int success` 0 0 for valid 1 for invalid

5 Conclusion

Level Files are also gzip'ed using zlib.h. Thus they also contain that level of compression.