41 Leveraging Plausibility Orderings to Achieve Extremely Efficient Data Compression

Jeff Rollason

41.1 Introduction

Data compression is often blind to the meaning of the data and instead depends on detecting pure, context-free patterns. It is possible, however, to exploit the structured nature of the data to achieve extremely efficient compression. In this chapter, we show how we at AI factory exploited this idea for our mobile Klondike Solitaire game *Solitaire Free* (Solitaire Free 2016), using a very simple approach to achieve compression far in excess of that provided by conventional methods—with final compressed data that were about 100× smaller than the raw data! Given that this was destined for a mobile app, this was a game changer for our team.

The underlying basis of our approach is to exploit the plausibility ordering, an attribute that has been crucial in-game AI for the success of programs such as AlphaGo (Deepmind 2016). We show here how this ordering can be used to yield other benefits, beyond just better gameplay.

41.2 The Core Requirement: A Klondike Solitaire App

Klondike Solitaire (Klondike 2016) is probably the world's most popular solitaire game. In order to build a successful mobile version of the game, we needed to provide the player with a variety of solvable puzzles, ranging from easy to very hard.

By estimation some 15% of Klondike puzzles have no solution (depending on the rule set). The principles needed to solve Klondike are well known but unfortunately not fast enough to randomly create a reliably graded puzzle on-the-fly. Puzzles created this way would tend to be of poor quality and most often unsolvable. Therefore we needed a library of graded puzzles, created in advance.

41.3 Assessing the Space Requirement

Given that runtime puzzle generation was not an option, we set out to create a database of graded puzzles for Klondike. We wanted a puzzle set for each of three difficulty levels. Each puzzle set consists of two rule sets, and each rule set should have 2000 puzzles, so we need to store the solutions for `3 * 2*2000` puzzles. The solutions vary in length, but the longest of them require 198 moves, and the representation for a single Klondike move takes 8 bytes. Thus, the simplest (and worst) solution would require `3 * 2*2000 * 198 * 8` bytes, or over 19 MB just for the puzzles! This is for a mobile app for a consumer market where users fill up their phones with apps, music, and photos. Space is a highly coveted resource. Most of AI Factory's mobile apps are nowhere near this size and they have game code and artwork as well, so clearly this is not an acceptable solution.

Of course this assumes that all solutions require the worst case size of move record, which obviously could be compacted into variable length records. Doing this gives an average record size of 125 instead of 198, which would reduce the overall size to `3 * 2*2000 * 125 * 8 = 12,000,000` bytes, but that is still too much.

41.4 Cutting the Space Requirements, Step 1: Bit Packing

As a first step, we took a look at the move format and saw that the 8-bit bytes are wasteful. There are only 10 move types, so that can be stored in 4 bits. Klondike needs 52 possible cards, which requires 6 bits. The tableau destination needs only 3 bits, and so on. Packing the data more efficiently cut the move record from 8 bytes down to 3, which results in a total size of 4,500,000 bytes. This was looking better, but 4.5 Mb is still quite a bit bigger than the total size of our smallest released product. We can do better…

41.5 Cutting the Space Requirements, Step 2: Table Lookup

The next idea was to replace the coded moves with an index that gives the number of the move in the list of legal moves at that point in the game. This means that the record cannot be decoded without the actual game engine to determine what the move ordering is, as we have reduced a move to a simple index, but this index has no meaning unless we know the order in which the moves are generated in game. This makes the record context sensitive but gives a radical improvement. Each move can now be stored in a single byte, reducing the size to `3 * 2*2000 * 125 * 1 = 1,500,000` bytes.

At this point we might have considered stopping as this is about half of the size of our smallest mobile app, although note that this is just one table, not the complete app. Regardless, we can still do better…

41.6 Cutting the Space Requirements, Step 3: Plausibility + Morse/Huffman Encoding

If we had a strict ordering of the moves by expected quality (a *plausibility ordering*) we could then take advantage of the fact that the distribution of possible move list indices would not be random but rather highly skewed, so that the right move for the solution is much more likely to be near the beginning of the list than near the end. In other words, the first move in the moves list would be expected to be the most common, the second somewhat less common, and so on until we get to the least common move at the end of the list. This would allow us to use a Morse Code-like encoding (Morse 2016) such that the most common index (i.e., 1) would be the equivalent of the letter "E" in Morse, with a single bit representing it, and later moves in the list would take up to 5 bits. This is, of course, essentially a Huffman encoding (Huffman 2016), but we have chosen to express it in terms of Morse Code (which somewhat predates Huffman) as that is a simpler concept with a wider common understanding.

A Huffman encoding could result in a very high level of compression, with an estimated four moves per byte and an overall size of 380 Kb. This would be very small, but a better (and also simpler) solution is possible…

41.7 Cutting the Space Requirements, Step 4: Plausibility + Run-Length Encoding

Examination of the plausibility orderings for the actual solutions showed that the orderings' first move was right most of the time. This offers a better option: To use Run-Length Encoding (Run-length 2016) on the move lists. With this approach a series of nine zeros (i.e., 9 moves where the first move on the list is correct) could be reduced to a single byte of the form `0 × 89` or `128 + 9`. Such was the dominance of these first moves that other moves could be left as-is, occupying 7 bits per index.

With this approach, the entire set could be reduced to just 192,160 bytes—and this included the start-up 6 bytes needed for each solution, the random seed (needed to define the card deal), the length of record, and the number of moves. This is storing moves at an astonishing 12.8 moves per byte, ignoring the headers, and 7.8 moves per byte after header overheads. This is almost 100× smaller than the uncompressed data, and makes the point very strongly that plausibility ordering can have a dramatic impact on game database compression.

41.8 An Example Game Record and the Compressed Record

The impact of this method is most easily appreciated by examining a sample solution record. Table 41.1 is the very first game in the database, minus the 6-byte header needed for each solution. This complete 127 move game compresses to the following 3 byte sequence:

```
197,1,185
```

This is dramatically successful, although this particular example is a simple puzzle and thus offers a better chance of high compression than our more complex puzzles do. Nevertheless, it still seems quite remarkable!

41.9 Beyond Data Compression: Plausibility Ordering as an AI Tool

Generation of a good plausibility ordering, in which the early moves are the most likely, is a critical topic. This point was illustrated by the recent success of AlphaGo. Almost certainly key to AlphaGo's success was its capacity to pick an Expert chosen move in the first move ordering 57% of the time, compared to a previous best of 44% by other groups. If this margin had been only, say, 52% then it is quite likely that the human world Go champion, Lee Sedol, would have easily won the match.

To do some simple calculations, consider the impact on a 6-ply sequence: at 57% the chance of taking the "right" line on a rollout of six moves is a healthy 3.4%, but at 44% this figure is a more modest 0.72%. When you project this out to the 20 plies that AlphaGo searched, the divergence in performance is more dramatic: 57% gives you about a 200-fold better chance of hitting the best line the first time.

This feeds back into our chapter in *Game AI Pro* 2, "Interest Search—A Faster Minimax" (Rollason 2015). Again the key to the success of the product *Shotest* (Japanese Chess) was its capacity to get the right move, or at least good moves, to the top of the moves list, in a game with a serious issue with combinatorial search explosions. A good plausibility ordering, combined with the interest search mechanism, allowed *Shotest* to quickly enter the top rankings in Computer Shogi, even though it was authored by a programmer who played a poor game of Shogi! Even a slightly better ordering resulted in a much more efficient and effective tree search.

It is worth pointing out that plausibility ordering, as used in tree search, is not what you would exclusively use to pick a move to play. Its purpose is to provide candidates to explore in a search, so it can afford to make colossal mistakes in its first choice as long as these mistakes are uncommon. Put simply, as long as the search has a high probability of a good choice, a low probability of a terrible choice may have a very limited negative impact. This makes plausibility ordering much easier, since you need not put in the much

Table 41.1 First Solution in the Klondike Puzzle Database

01.	T♣	Tabl->Tabl 5->1 (From 5)
02	4♠	Draw
03.	K♣	Draw
04.	8♣	Draw
05.	A♣	Draw
06.	4♦	Draw
07.	T♦	Waste->Tabl->7
08.	K♥	Draw
09.	Q♥	Draw
10.	2♠	Waste->Tabl->2
11.	9♣	Draw
12.	?♥	Re-cycle
13.	4♠	Draw
14.	K♣	Draw
15.	9♠	Waste->Tabl->7
16.	8♦	Tabl->Tabl 3->7 (From 3)
17.	7♦	Tabl->Tabl 5->3 (From 4)
18.	7♣	Tabl->Tabl 5->7 (From 3)
19.	Q♠	Tabl->Tabl 5->4 (From 2)
20.	J♦	Tabl->Tabl 1->4 (From 1)
21.	8♣	Draw
22.	A♣	Draw
23.	4♦	Draw
24.	K♥	Waste->Tabl->1
25.	2♦	Draw
26.	K♠	Draw
27.	Q♣	Waste->Tabl->1
28.	T♠	Draw
29.	?♥	Re-cycle
30.	4♠	Draw
31.	6♦	Waste->Tabl->7
32.	K♣	Draw
33.	9♥	Draw
34.	A♣	Waste->Foundation->1
35.	A♦	Draw
36.	J♥	Draw
37.	Q♥	Draw
38.	T♠	Draw
39.	?♥	Re-cycle
40.	4♠	Draw
41.	5♣	Draw
42.	9♥	Waste->Tabl->4
43.	8♣	Waste->Tabl->4
44.	5♣	Waste->Tabl->7
45.	4♥	Tabl->Tabl 5->7 (From 1)
46.	K♣	Waste->Tabl->5
47.	Q♦	Draw
48.	7♥	Waste->Tabl->4
49.	A♦	Waste->Foundation->2
50.	Q♦	Waste->Tabl->5
51.	4♦	Draw
52.	2♦	Waste->Foundation->2
53.	J♥	Waste->Tabl->1
54.	8♥	Draw
55.	9♣	Draw
56.	T♠	Waste->Tabl->1
57.	?♥	Re-cycle
58.	J♣	Tabl->Tabl 7->5 (From 7)
59.	3♦	Tabl->Foundation 7->2
60.	A♥	Tabl->Foundation 7->3
61.	5♥	Tabl->Tabl 6->7 (From 6)
62.	2♣	Tabl->Foundation 6->1
63.	2♥	Tabl->Foundation 6->3
64.	3♠	Tabl->Tabl 6->5 (From 3)
65.	6♣	Tabl->Tabl 7->4 (From 4)
66.	T♥	Tabl->Tabl 6->7 (From 2)
67.	6♠	Tabl->Tabl 6->3 (From 1)
68.	K♦	Tabl->Tabl 4->6 (From 4)
69.	3♣	Tabl->Foundation 4->1
70.	3♥	Tabl->Tabl 2->4 (From 2)
71.	5♦	Tabl->Tabl 2->3 (From 1)
72.	4♣	Tabl->Tabl 4->3 (From 2)
73.	4♠	Draw
74.	4♦	Waste->Foundation->2
75.	8♥	Draw
76.	K♠	Waste->Tabl->2
77.	Q♥	Waste->Tabl->2
78.	J♠	Tabl->Tabl 7->2 (From 3)
79.	A♠	Tabl->Foundation 7->4
80.	2♠	Tabl->Foundation 3->4
81.	3♥	Tabl->Foundation 3->3
82.	4♣	Tabl->Foundation 3->1
83.	5♦	Tabl->Foundation 3->2
84.	3♠	Tabl->Foundation 5->4
85.	4♥	Tabl->Foundation 5->3
86.	5♣	Tabl->Foundation 5->1
87.	6♦	Tabl->Foundation 5->2
88.	5♥	Tabl->Foundation 6->3
89.	6♣	Tabl->Foundation 6->1
90.	7♣	Tabl->Foundation 5->1
91.	8♠	Tabl->Tabl 3->7 (From 2)
92.	9♦	Tabl->Tabl 7->1 (From 1)

(*Continued*)

Table 41.1 (*Continued*) First Solution in the Klondike Puzzle Database

93. 7♠ Tabl->Tabl 3->5 (From 1)	111. 8♣ Tabl->Foundation 6->1
94. 9♣ Draw	112. 9♣ Tabl->Foundation 2->1
95. 9♣ Waste->Tabl->2	113. 9♥ Tabl->Foundation 6->3
96. 8♥ Waste->Tabl->2	114. T♥ Tabl->Foundation 2->3
97. 6♥ Waste->Foundation->3	115. J♥ Tabl->Foundation 1->3
98. 4♠ Waste->Foundation->4	116. J♠ Tabl->Foundation 2->4
99. 5♠ Tabl->Foundation 4->4	117. Q♥ Tabl->Foundation 2->3
100. 6♠ Tabl->Foundation 1->4	118. T♣ Tabl->Foundation 6->1
101. 7♦ Tabl->Foundation 1->2	119. J♣ Tabl->Foundation 5->1
102. 7♠ Tabl->Foundation 5->4	120. Q♣ Tabl->Foundation 1->1
103. 8♠ Tabl->Foundation 1->4	121. K♥ Tabl->Foundation 1->3
104. 8♦ Tabl->Foundation 5->2	122. J♦ Tabl->Foundation 6->2
105. 9♦ Tabl->Foundation 1->2	123. Q♦ Tabl->Foundation 5->2
106. 9♠ Tabl->Foundation 5->4	124. K♣ Tabl->Foundation 5->1
107. T♠ Tabl->Foundation 1->4	125. Q♠ Tabl->Foundation 6->4
108. T♦ Tabl->Foundation 5->2	126. K♠ Tabl->Foundation 2->4
109. 7♥ Tabl->Foundation 6->3	127. K♦ Tabl->Foundation 6->2
110. 8♥ Tabl->Foundation 2->3	

larger effort to avoid rare poor choices. The dynamic of this is significantly different from the needs of the linear evaluation function that might ultimately be responsible for the chosen move.

41.10 Conclusion

This case study shows that plausibility ordering, where better moves are mostly ordered at the top, offers a way to impose a highly structured skew on the data, making it amenable to compression far in excess of what is possible from context-free data compression. Huffman encoding would have allowed the original database of raw solutions to be compressed, but this would have depended on detecting moves common to all solutions. This moves list would have included a relatively vast number of possible moves. However, reclassifying the solutions, not by move content, but by chance of the move being correct, offers a substantially more efficient structure for compression. The complexity of move structure is then completely removed from the database and replaced by a minimalized move index.

In this case our plausibility ordering skewed very heavily to the first move in the list, but another inferior ordering might also perform very well, utilizing either a Huffman encoding or a modified Run-Length Encoding. To get the optimum compression for the latter would probably need some simple hand coding. If the second move had a high chance of being correct then the byte structure might use the top 2 bits, leaving 6 bits for uncompressed moves. This would allow the three top moves to use Run-Length Encoding. If a move sequence of the same positions exceeded 6 bits then it could simply be split into an additional byte for the rare overflow. In our encoding, for example, we might have done better by encoding the top three moves instead of just the top move—but we had already achieved plenty, and time is money!

Of course in more complex distributions the run-length option might simply not work, but the Huffman encoding would almost certainly work in these cases.

References

AlphaGo. https://deepmind.com/alpha-go (accessed June 18, 2016).

Rollason, J. 2015. Interest search—A faster minimax. In *Game AI Pro 2*, ed, S. Rabin. Boca Raton, FL: CRC Press, pp. 255–264.

Klondike (solitaire). https://en.wikipedia.org/wiki/Klondike_(solitaire) (accessed June 18, 2016).

Morse code. https://en.wikipedia.org/wiki/Morse_code (accessed June 18, 2016).

Run-Length Encoding. https://en.wikipedia.org/wiki/Run-length_encoding (accessed June 18, 2016).

Huffman coding. https://en.wikipedia.org/wiki/Huffman_coding (accessed June 18, 2016).

Solitaire Free. https://play.google.com/store/apps/details?id=uk.co.aifactory.solitairefree (accessed June 18, 2016).