

# Programmer Puzzle: Encoding a chess board state throughout a game

Asked 15 years ago   Modified 5 years, 7 months ago   Viewed 58k times

---



100



Not strictly a question, more of a puzzle...

Over the years, I've been involved in a few technical interviews of new employees. Other than asking the standard "do you know X technology" questions, I've also tried to get a feel for how they approach problems.

Typically, I'd send them the question by email the day before the interview, and expect them to come up with a solution by the following day.

Often the results would be quite interesting - wrong, but interesting - and the person would still get my recommendation if they could explain why they took a particular approach.

So I thought I'd throw one of my questions out there for the Stack Overflow audience.

Question: **What is the most space-efficient way you can think of to encode the state of a chess game (or subset thereof)?** That is, given a chess board with the pieces arranged legally, encode both this initial state

**and all subsequent legal moves taken by the players in the game.**

No code required for the answer, just a description of the algorithm you would use.

EDIT: As one of the posters has pointed out, I didn't consider the time interval between moves. Feel free to account for that too as an optional extra :)

EDIT2: Just for additional clarification... Remember, the encoder/decoder is rule-aware. The only things that really need to be stored are the player's choices - anything else can be assumed to be known by the encoder/decoder.

EDIT3: It's going to be difficult to pick a winner here :)  
Lots of great answers!

algorithm

language-agnostic

puzzle

chess

Share

edited Dec 3, 2009 at 8:56

Improve this question

Follow

community wiki

5 revs, 2 users 100%

Andrew Rollings

- 
- 4    Isn't the initial state of a chess game well-defined? Why does it have to be encoded? I think it should be enough to encode

the diffs between each turnn (=moves), only. – [tanascius](#) Dec 2, 2009 at 8:59

---

6 to be strict, you'll also have to encode all the past positions, because if the same position appears three times it's a draw [en.wikipedia.org/wiki/Threefold\\_repetition](http://en.wikipedia.org/wiki/Threefold_repetition) – [flybywire](#) Dec 2, 2009 at 9:15

---

4 Suggestion: make this a real contest where people submit their entries as programs. A program would take a chess game as its input (you could define some basic, human-readable, non-optimized format for this) and would output the compressed game. Then, with a parameter, it would take the compressed game and regenerate the original input which would have to match. – [Vilx-](#) Dec 2, 2009 at 9:32

---

2 I'd reply "storage is cheap. Store is in a human readable form and then zip it if you have a billion of 'em"  
– [No Refunds No Returns](#) Dec 3, 2009 at 6:22

---

2 More to the point, it would demonstrate that you can't follow instructions... Even the most ubercoder needs to follow instructions at some point. I've run into situations where I've been told to implement something in a certain way, even though I've thought (and said) it was a stupid implementation, only to be left with egg on my face when it turned out that there was a very good reason (that I didn't know or comprehend) to have it implemented that way.  
– [Andrew Rollings](#) Dec 3, 2009 at 8:52

---

31 Answers

Sorted by:

Highest score (default)



1

2

Next



**Update:** I liked this topic so much I wrote [Programming Puzzles, Chess Positions and Huffman Coding](#). If you

132



read through this I've determined that the *only* way to store a complete game state is by storing a complete list of moves. Read on for why. So I use a slightly simplified version of the problem for piece layout.

## The Problem

This image illustrates the starting Chess position. Chess occurs on an 8x8 board with each player starting with an identical set of 16 pieces consisting of 8 pawns, 2 rooks, 2 knights, 2 bishops, 1 queen and 1 king as illustrated here:



Positions are generally recorded as a letter for the column followed by the number for the row so White's queen is at d1. Moves are most often stored in [algebraic notation](#), which is unambiguous and generally only

specifies the minimal information necessary. Consider this opening:

1. e4 e5
2. Nf3 Nc6
3. ...

which translates to:

1. White moves king's pawn from e2 to e4 (it is the only piece that can get to e4 hence "e4");
2. Black moves the king's pawn from e7 to e5;
3. White moves the knight (N) to f3;
4. Black moves the knight to c6.
5. ...

The board looks like this:



An important ability for any programmer is to be able to *correctly and unambiguously specify the problem*.

So what's missing or ambiguous? A lot as it turns out.

## Board State vs Game State

The first thing you need to determine is whether you're storing the state of a game or the position of pieces on the board. Encoding simply the positions of the pieces is one thing but the problem says "all subsequent legal moves". The problem also says nothing about knowing the moves up to this point. That's actually a problem as I'll explain.

## Castling

The game has proceeded as follows:

1. e4 e5
2. Nf3 Nc6
3. Bb5 a6
4. Ba4 Bc5

The board looks as follows:



White has the option of [castling](#). Part of the requirements for this are that the king and the relevant rook can never have moved, so whether the king or either rook of each side has moved will need to be stored. Obviously if they aren't on their starting positions, they have moved otherwise it needs to be specified.

There are several strategies that can be used for dealing with this problem.

Firstly, we could store an extra 6 bits of information (1 for each rook and king) to indicate whether that piece had moved. We could streamline this by only storing a bit for one of these six squares if the right piece happens to be in it. Alternatively we could treat each unmoved piece as another piece type so instead of 6 piece types on each side (pawn, rook, knight, bishop, queen and king) there are 8 (adding unmoved rook and unmoved king).

# En Passant

Another peculiar and often-neglected rule in Chess is [En Passant](#).



The game has progressed.

1. e4 e5
2. Nf3 Nc6
3. Bb5 a6
4. Ba4 Bc5
5. O-O b5
6. Bb3 b4
7. c4

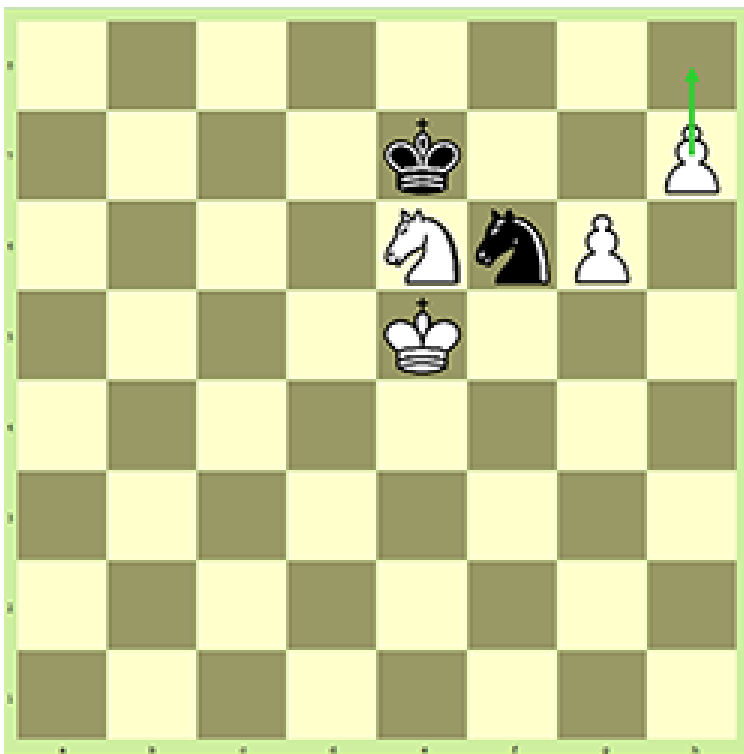
Black's pawn on b4 now has the option of moving his pawn on b4 to c3 taking the White pawn on c4. This only



happens on the first opportunity meaning if Black passes on the option now he can't take it next move. So we need to store this.

If we know the previous move we can definitely answer if En Passant is possible. Alternatively we can store whether each pawn on its 4th rank has just moved there with a double move forward. Or we can look at each possible En Passant position on the board and have a flag to indicate whether its possible or not.

## Promotion



It is White's move. If White moves his pawn on h7 to h8 it can be promoted to any other piece (but not the king). 99% of the time it is promoted to a Queen but sometimes it isn't, typically because that may force a stalemate when otherwise you'd win. This is written as:

56. h8=Q

This is important in our problem because it means we can't count on there being a fixed number of pieces on each side. It is entirely possible (but incredibly unlikely) for one side to end up with 9 queens, 10 rooks, 10 bishops or 10 knights if all 8 pawns get promoted.

## Stalemate

When in a position from which you cannot win your best tactic is to try for a [stalemate](#). The most likely variant is where you cannot make a legal move (usually because any move when put your king in check). In this case you can claim a draw. This one is easy to cater for.

The second variant is by [threefold repetition](#). If the same board position occurs three times in a game (or will occur a third time on the next move), a draw can be claimed. The positions need not occur in any particular order (meaning it doesn't have to be the same sequence of moves repeated three times). This one greatly complicates the problem because you have to remember every previous board position. ***If this is a requirement of the problem the only possible solution to the problem is to store every previous move.***

Lastly, there is the [fifty move rule](#). A player can claim a draw if no pawn has moved and no piece has been taken in the previous fifty consecutive moves so we would need

to store how many moves since a pawn was moved or a piece taken (the latest of the two. This requires 6 bits (0-63).

## Whose Turn Is It?

Of course we also need to know whose turn it is and this is a single bit of information.

## Two Problems

Because of the stalemate case, the only feasible or sensible way to store the game state is to store all the moves that led to this position. I'll tackle that one problem. The board state problem will be simplified to this: *store the current position of all pieces on the board ignoring castling, en passant, stalemate conditions and whose turn it is.*

Piece layout can be broadly handled in one of two ways: by storing the contents of each square or by storing the position of each piece.

## Simple Contents

There are six piece types (pawn, rook, knight, bishop, queen and king). Each piece can be White or Black so a

square may contain one of 12 possible pieces or it may be empty so there are 13 possibilities. 13 can be stored in 4 bits (0-15) So the simplest solution is to store 4 bits for each square times 64 squares or 256 bits of information.

The advantage of this method is that manipulation is *incredibly* easy and fast. This could even be extended by adding 3 more possibilities without increasing the storage requirements: a pawn that has moved 2 spaces on the last turn, a king that hasn't moved and a rook that hasn't moved, which will cater for a lot of previously mentioned issues.

But we can do better.

## Base 13 Encoding

It is often helpful to think of the board position as a very large number. This is often done in computer science. For example, the [halting problem](#) treats a computer program (rightly) as a large number.

The first solution treats the position as a 64 digit base 16 number but as demonstrated there is redundancy in this information (being the 3 unused possibilities per "digit") so we can reduce the number space to 64 base 13 digits. Of course this can't be done as efficiently as base 16 can but it will save on storage requirements (and minimizing storage space is our goal).

In base 10 the number 234 is equivalent to  $2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$ .

In base 16 the number 0xA50 is equivalent to  $10 \times 16^2 + 5 \times 16^1 + 0 \times 16^0 = 2640$  (decimal).

So we can encode our position as  $p_0 \times 13^{63} + p_1 \times 13^{62} + \dots + p_{63} \times 13^0$  where  $p_i$  represents the contents of square  $i$ .

$2^{256}$  equals approximately  $1.16e77$ .  $13^{64}$  equals approximately  $1.96e71$ , which requires 237 bits of storage space. That saving of a mere 7.5% comes at a cost of **significantly** increased manipulation costs.

## Variable Base Encoding

In legal boards certain pieces can't appear in certain squares. For example, pawns cannot occur at in the first or eighth ranks, reducing the possibilities for those squares to 11. That reduces the possible boards to  $11^{16} \times 13^{48} = 1.35e70$  (approximately), requiring 233 bits of storage space.

Actually encoding and decoding such values to and from decimal (or binary) is a little more convoluted but it can be done reliably and is left as an exercise to the reader.

## Variable Width Alphabets

The previous two methods can both be described as *fixed-width alphabetic encoding*. Each of the 11, 13 or 16

members of the alphabet is substituted for another value. Each “character” is the same width but the efficiency can be improved when you consider that each character is not equally likely.

## MORSE CODE

A	• —	N	— •	1	• — — — —
B	— • • •	O	— — —	2	• • — — —
C	— • • •	P	• — — •	3	• • • — —
D	— • •	Q	— — • —	4	• • • • —
E	•	R	• — •	5	• • • • •
F	• • — •	S	• • •	6	— • • • •
G	— — •	T	—	7	— — — • •
H	• • • •	U	• • —	8	— — — — • •
I	• •	V	• • • —	9	— — — — •
J	• — — — —	W	• — — —	0	— — — — —
K	— • —	X	— • • —		
L	• — • •	Y	— • — — —		
M	— —	Z	— — • •		

Consider [Morse code](#) (pictured above). Characters in a message are encoded as a sequence of dashes and dots. Those dashes and dots are transferred over radio (typically) with a pause between them to delimit them.

Notice how the letter E ([the most common letter in English](#)) is a single dot, the shortest possible sequence, whereas Z (the least frequent) is two dashes and two beeps.

Such a scheme can significantly reduce the size of an *expected* message but comes at the cost of increasing the size of a random character sequence.

It should be noted that Morse code has another inbuilt feature: dashes are as long as three dots so the above code is created with this in mind to minimize the use of

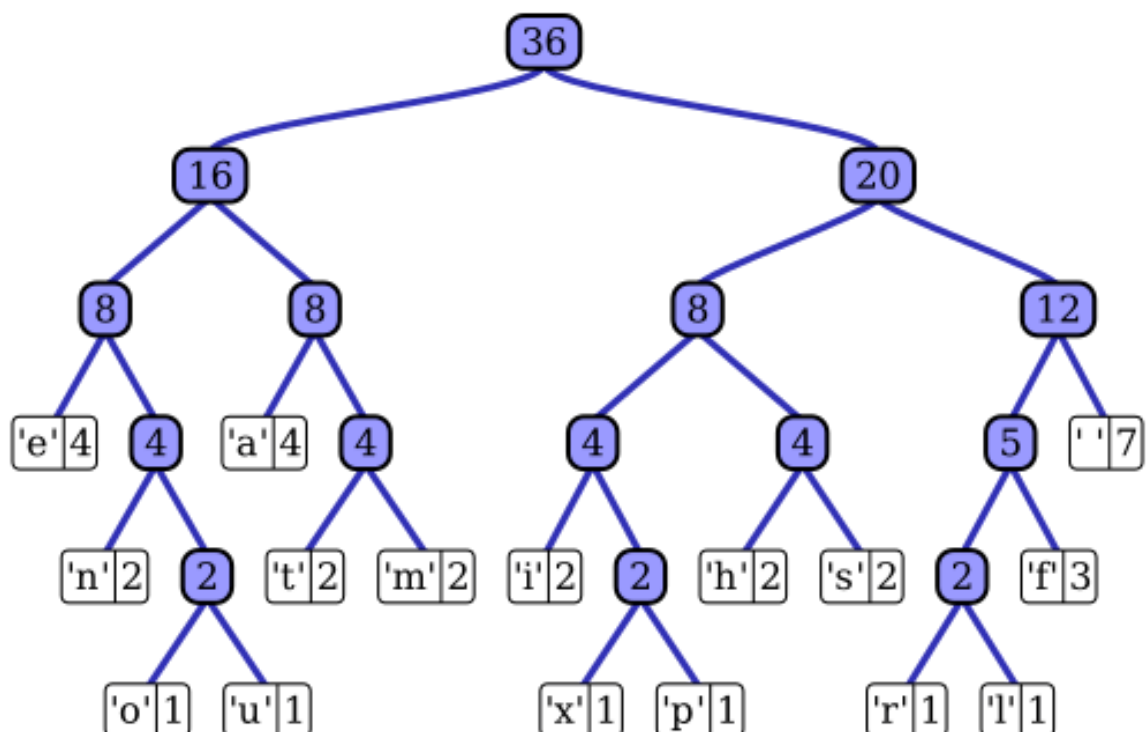
dashes. Since 1s and 0s (our building blocks) don't have this problem, it's not a feature we need to replicate.

Lastly, there are two kinds of rests in Morse code. A short rest (the length of a dot) is used to distinguish between dots and dashes. A longer gap (the length of a dash) is used to delimit characters.

So how does this apply to our problem?

## Huffman Coding

There is an algorithm for dealing with variable length codes called [Huffman coding](#). Huffman coding creates a variable length code substitution, typically uses expected frequency of the symbols to assign shorter values to the more common symbols.



In the above tree, the letter E is encoded as 000 (or left-left-left) and S is 1011. It should be clear that this encoding scheme is *unambiguous*.

This is an important distinction from Morse code. Morse code has the character separator so it can do otherwise ambiguous substitution (eg 4 dots can be H or 2 Is) but we only have 1s and 0s so we choose an unambiguous substitution instead.

Below is a simple implementation:

```
private static class Node {
    private final Node left;
    private final Node right;
    private final String label;
    private final int weight;

    private Node(String label, int weight) {
        this.left = null;
        this.right = null;
        this.label = label;
        this.weight = weight;
    }

    public Node(Node left, Node right) {
        this.left = left;
        this.right = right;
        label = "";
        weight = left.weight + right.weight;
    }

    public boolean isLeaf() { return left == null && right == null; }

    public Node getLeft() { return left; }

    public Node getRight() { return right; }

    public String getLabel() { return label; }
```



```

    public int getWeight() { return weight; }
}

```

with static data:

```

private final static List<string> COLOURS;
private final static Map<string, integer> WEIGHTS;

static {
    List<string> list = new ArrayList<string>();
    list.add("White");
    list.add("Black");
    COLOURS = Collections.unmodifiableList(list);
    Map<string, integer> map = new HashMap<string, integer>();
    for (String colour : COLOURS) {
        map.put(colour + " " + "King", 1);
        map.put(colour + " " + "Queen", 1);
        map.put(colour + " " + "Rook", 2);
        map.put(colour + " " + "Knight", 2);
        map.put(colour + " " + "Bishop", 2);
        map.put(colour + " " + "Pawn", 8);
    }
    map.put("Empty", 32);
    WEIGHTS = Collections.unmodifiableMap(map);
}

```

and:

```

private static class WeightComparator implements Comparator<Node> {
    @Override
    public int compare(Node o1, Node o2) {
        if (o1.getWeight() == o2.getWeight()) {
            return 0;
        } else {
            return o1.getWeight() < o2.getWeight() ? -1 : 1;
        }
    }
}

```

```

private static class PathComparator implements Comparable<Path> {
    @Override
    public int compare(String o1, String o2) {
        if (o1 == null) {
            return o2 == null ? 0 : -1;
        } else if (o2 == null) {
            return 1;
        } else {
            int length1 = o1.length();
            int length2 = o2.length();
            if (length1 == length2) {
                return o1.compareTo(o2);
            } else {
                return length1 < length2 ? -1 : 1;
            }
        }
    }
}

public static void main(String args[]) {
    PriorityQueue<node> queue = new PriorityQueue<node>(
        new WeightComparator());
    for (Map.Entry<string, integer> entry : WEIGHTS.entrySet()) {
        queue.add(new Node(entry.getKey(), entry.getValue()));
    }
    while (queue.size() > 1) {
        Node first = queue.poll();
        Node second = queue.poll();
        queue.add(new Node(first, second));
    }
    Map<string, node> nodes = new TreeMap<string, node>(
        new WeightComparator());
    addLeaves(nodes, queue.peek(), "&quot;&quot;");
    for (Map.Entry<string, node> entry : nodes.entrySet()) {
        System.out.printf("%s %s\n", entry.getKey(), entry.getValue());
    }
}

public static void addLeaves(Map<string, node> nodes,
    {
        if (node != null) {
            addLeaves(nodes, node.getLeft(), prefix + "0");
            addLeaves(nodes, node.getRight(), prefix + "1");
            if (node.isLeaf()) {
                nodes.put(prefix, node);
            }
        }
    }
}

```

```
}  
}  
}
```

One possible output is:

	White	Black
Empty	0	
Pawn	110	100
Rook	11111	11110
Knight	10110	10101
Bishop	10100	11100
Queen	111010	111011
King	101110	101111

For a starting position this equates to  $32 \times 1 + 16 \times 3 + 12 \times 5 + 4 \times 6 = 164$  bits.

## State Difference

Another possible approach is to combine the very first approach with Huffman coding. This is based on the assumption that most expected Chess boards (rather than randomly generated ones) are more likely than not to, at least in part, resemble a starting position.

So what you do is XOR the 256 bit current board position with a 256 bit starting position and then encode that (using Huffman coding or, say, some method of [run length encoding](#)). Obviously this will be very efficient to start with (64 0s probably corresponding to 64 bits) but increase in storage required as the game progresses.

# Piece Position

As mentioned, another way of attacking this problem is to instead store the position of each piece a player has. This works particularly well with endgame positions where most squares will be empty (but in the Huffman coding approach empty squares only use 1 bit anyway).

Each side will have a king and 0-15 other pieces. Because of promotion the exact make up of those pieces can vary enough that you can't assume the numbers based on the starting positions are maxima.

The logical way to divide this up is store a Position consisting of two Sides (White and Black). Each Side has:

- A king: 6 bits for the location;
- Has pawns: 1 (yes), 0 (no);
- If yes, number of pawns: 3 bits ( $0-7+1 = 1-8$ );
- If yes, the location of each pawn is encoded: 45 bits (see below);
- Number of non-pawns: 4 bits (0-15);
- For each piece: type (2 bits for queen, rook, knight, bishop) and location (6 bits)

As for the pawn location, the pawns can only be on 48 possible squares (not 64 like the others). As such, it is better not to waste the extra 16 values that using 6 bits per pawn would use. So if you have 8 pawns there are

$48^8$  possibilities, equalling 28,179,280,429,056. You need 45 bits to encode that many values.

That's 105 bits per side or 210 bits total. The starting position is the worst case for this method however and it will get substantially better as you remove pieces.

It should be pointed out that there are less than  $48^8$  possibilities because the pawns can't all be in the same square. The first has 48 possibilities, the second 47 and so on.  $48 \times 47 \times \dots \times 41 = 1.52e13 = 44$  bits storage.

You can further improve this by eliminating the squares that are occupied by other pieces (including the other side) so you could first place the white non-pawns then the black non-pawns, then the white pawns and lastly the black pawns. On a starting position this reduces the storage requirements to 44 bits for White and 42 bits for Black.

## Combined Approaches

Another possible optimization is that each of these approaches has its strength and weaknesses. You could, say, pick the best 4 and then encode a scheme selector in the first two bits and then the scheme-specific storage after that.

With the overhead that small, this will by far be the best approach.

# Game State

I return to the problem of storing a *game* rather than a *position*. Because of the threefold repetition we have to store the list of moves that have occurred to this point.

## Annotations

One thing you have to determine is are you simply storing a list of moves or are you annotating the game? Chess games are often annotated, for example:

17. Bb5!! Nc4?

White's move is marked by two exclamation points as brilliant whereas Black's is viewed as a mistake. See [Chess punctuation](#).

Additionally you could also need to store free text as the moves are described.

I am assuming that the moves are sufficient so there will be no annotations.

## Algebraic Notation

We could simply store the the text of the move here ("e4", "Bxb5", etc). Including a terminating byte you're looking at

about 6 bytes (48 bits) per move (worst case). That's not particularly efficient.

The second thing to try is to store the starting location (6 bits) and end location (6 bits) so 12 bits per move. That is significantly better.

Alternatively we can determine all the legal moves from the current position in a predictable and deterministic way and state which we've chosen. This then goes back to the variable base encoding mentioned above. White and Black have 20 possible moves each on their first move, more on the second and so on.

## Conclusion

There is no absolutely right answer to this question. There are many possible approaches of which the above are just a few.


What I like about this and similar problems is that it demands abilities important to any programmer like considering the usage pattern, accurately determining requirements and thinking about corner cases.

*Chess positions taken as screenshots from [Chess Position Trainer](#).*

community wiki

14 revs, 8 users 76%

cletus

- 
- 3 and gzip the result afterwards (if the headers won't increase the result) ;^) – [Toad](#) Dec 2, 2009 at 8:23
- 
- 5 Good post. Small correction: castling requires 4 bits, one for each way of castling (white and black, kingside and queenside), because the rooks might have moved and then moved back also. Somewhat more important: you should probably include whose move it is. => – [A. Rex](#) Dec 2, 2009 at 9:40
- 
- 10 As for promoting to a knight, I've done that once. Really wild situation--he was one move from mating me, I couldn't stop it. He had ignored my pawn because while it would promote it would be one move late. I wish I had my camera when I promoted to a knight instead and mated him!  
– [Loren Pechtel](#) Dec 3, 2009 at 4:40
- 
- 2 I'm surprised your article didn't mention [FEN][1], which handles castling, en passant availability, etc. [1]  
[en.wikipedia.org/wiki/FEN](http://en.wikipedia.org/wiki/FEN) – [Ross](#) Jan 1, 2010 at 5:16 
- 
- 1 Even to fully support the 50 moves rule and three repetitions rule, it is only strictly necessary to represent the board contents just after the last time a pawn was moved or a piece captured (with castling and en passant rights), plus all moves since that position. It is not necessary to record the entire history for this purpose. – [wberry](#) Apr 7, 2014 at 19:09
-





It's best just to store chess games in a human-readable, standard format.

48



The [Portable Game Notation](#) assumes a standard starting position (although it [doesn't have to](#)) and just lists the moves, turn by turn. A compact, human-readable, standard format.



E.g.

```
[Event "F/S Return Match"]
[Site "Belgrade, Serbia Yugoslavia|JUG"]
[Date "1992.11.04"]
[Round "29"]
[White "Fischer, Robert J."]
[Black "Spasky, Boris V."]
[Result "1/2-1/2"]
```

```
1. e4 e5 2. Nf3 Nc6 3. Bb5 {This opening is called
the Ruy Lopez.} 3... a6
4. Ba4 Nf6 5. O-O Be7 6. Re1 b5 7. Bb3 d6 8. c3 O-
O 9. h3 Nb8 10. d4 Nbd7
11. c4 c6 12. cxb5 axb5 13. Nc3 Bb7 14. Bg5 b4 15.
Nb1 h6 16. Bh4 c5 17. dxe5
Nxe4 18. Bxe7 Qxe7 19. exd6 Qf6 20. Nbd2 Nxd6 21.
Nc4 Nxc4 22. Bxc4 Nb6
23. Ne5 Rae8 24. Bxf7+ Rxf7 25. Nxf7 Rxe1+ 26.
Qxe1 Kxf7 27. Qe3 Qg5 28. Qxg5
hxg5 29. b3 Ke6 30. a3 Kd6 31. axb4 cxb4 32. Ra5
Nd5 33. f3 Bc8 34. Kf2 Bf5
35. Ra7 g6 36. Ra6+ Kc5 37. Ke1 Nf4 38. g3 Nxh3
39. Kd2 Kb5 40. Rd6 Kc5 41. Ra6
Nf2 42. g4 Bd3 43. Re6 1/2-1/2
```

If you want to make it smaller, then [just zip it](#). Job done!

---

23 In my defence against the 2 downvotes this received: 1) It does what you want 2) It passes the [thedailywtf.com/articles/riddle-me-an-interview.aspx](http://thedailywtf.com/articles/riddle-me-an-interview.aspx) test: "...some of the folks who can solve these riddles are precisely the type of people you don't want as programmers. Would you want to work with the guy who builds a water-displacement scale/barge, taxis a 747 to the docks, and then weights the jumbo jet using that, instead of simply calling Boeing in the first place?" You don't hire someone who invents you a random encoding in interview, because they'll do it in their code as well. – Rob Grant Dec 2, 2009 at 11:04

---

1 Well, if I'm specifically asking them to solve a problem to get their problem solving technique, then you can assume I'll cover the other stuff with other questions...  
– Andrew Rollings Dec 2, 2009 at 11:11

---

7 @reinier: I'm not saying I'm completely clueless regarding information density problems (you misdiagnosed my reply as being an admission of incompetence). Surely you want to hire the person who codes to an existing data storage standard, and who recognises that using appropriate existing tools rather than rolling his/her own can be a good idea - "We invented The Wheel 2.0! It's now even rounder!" You definitely don't want to hire the person who thinks - bizarrely - that using library functions is a sign of weakness.  
– Rob Grant Dec 2, 2009 at 14:39

---

18 This would absolutely be my first answer to this question in an interview. You want to show that your first instinct is to look for a ready solution. If the interviewer tells you they want to hear what you can come up with on your own, then you

can go into a bit-packing solution. – [Bill the Lizard](#) Dec 3, 2009 at 6:02

---

- 2 I am with Robert on this one - existing solution is practical, human readable and compact enough. All are MAJOR feats compared to custom super packed solution with complicated algorithms to decode them. If it's about interview I would definitely also consider practical aspect! You'd be amazed how many times really smart people come up with hyper complicated impractical solutions. It's usually attributed to the fact that they can handle complexity in their head, but then - what about rest of us... – [MaR](#) Dec 3, 2009 at 12:20
- 



Great puzzle!

15



I see that most people are storing the position of each piece. How about taking a more simple-minded approach, and **storing the contents of each square**? That takes care of promotion and captured pieces automatically.



And it allows for **Huffman encoding**. Actually, the initial frequency of pieces on the board is nearly perfect for this: half of the squares are empty, half of the remaining squares are pawns, etcetera.

Considering the frequency of each piece, I constructed a [Huffman tree](#) on paper, which I won't repeat here. The result, where **c** stands for the colour (white = 0, black = 1):

- 0 for empty squares
- 1c0 for pawn

- 1c100 for rook
- 1c101 for knight
- 1c110 for bishop
- 1c1110 for queen
- 1c1111 for king

For the entire board in its initial situation, we have

- empty squares:  $32 * 1 \text{ bit} = 32 \text{ bits}$
- pawns:  $16 * 3 \text{ bits} = 48 \text{ bits}$
- rooks/knights/bishops:  $12 * 5 \text{ bits} = 60 \text{ bits}$
- queens/kings:  $4 * 6 \text{ bits} = 24 \text{ bits}$

Total: **164 bits** for the *initial* board state. Significantly less than the 235 bits of the currently highest voted answer. And it's only going to get smaller as the game progresses (except after a promotion).

I looked only at the position of the pieces on the board; additional state (whose turn, who has castled, en passant, repeating moves, etc.) will have to be encoded separately. Maybe another 16 bits at most, so **180 bits** for the entire game state. Possible optimizations:

- Leaving out the less frequent pieces, and storing their position separately. But that won't help... replacing king and queen by an empty square saves 5 bits, which are exactly the 5 bits you need to encode their position in another way.

- "No pawns on the back row" could easily be encoded by using a different Huffman table for the back rows, but I doubt it helps much. You'd probably still end up with the same Huffman tree.
- "One white, one black bishop" can be encoded by introducing extra symbols that don't have the **c** bit, which can then be deduced from the square that the bishop is on. (Pawns promoted to bishops disrupt this scheme...)
- Repetitions of empty squares could be run-length encoded by introducing extra symbols for, say, "2 empty squares in a row" and "4 empty squares in a row". But it is not so easy to estimate the frequency of those, and if you get it wrong, it's going to hurt rather than help.

Share Improve this answer

answered [Dec 2, 2009 at 9:47](#)

Follow

community wiki

[Thomas](#)

---

No pawns on the bank ranks does save a bit--you can chop bit #3 out of all the other patterns. Thus you will save one bit per piece actually on a bank rank. – [Loren Pechtel](#) Dec 3, 2009 at 4:43

- 
- 2 You can do a separate Huffman tree for each of the 64 squares, as some likely have some pieces more often than others. – [Claudiu](#) Dec 3, 2009 at 8:04
-



# The really big lookup table approach

9

**Position** - 18 bytes

[Estimated number of legal positions is](#)  $10^{43}$



Simply enumerate them all and the position can be stored in just 143 bits. 1 more bit is required to indicate which side is to play next



The enumeration is not practical of course, but this shows that at least 144 bits are required.

**Moves** - 1 byte

There are usually around 30-40 legal moves for each position but the number may be as high as 218. Let's enumerate all the legal moves for each position. Now each move can be encoded into one byte.

We still have plenty of room for special moves such as 0xFF to represent resigning.

Share Improve this answer

[edited Dec 3, 2009 at 5:54](#)

Follow

community wiki

[3 revs](#)

[gnibbler](#)

- 
- 3 Straight to the heart of the requirement "most space-efficient way you can think of to encode the state of a chess game" - Nothing is better to squash something than a dictionary, and that includes a fly. – [Andrew](#) Dec 3, 2009 at 8:50
-

- 
- 1 I found an interesting link on how long it would take to generate such a dictionary :)  
[ioannis.virtualcomposer2000.com/math/EveryChess.html](http://ioannis.virtualcomposer2000.com/math/EveryChess.html)  
– Andrew Rollings Dec 3, 2009 at 8:54
- 

- 1 Shannons estimate is a bit outdated :-) He did include neither promotions nor captures, which blow the number up by a fair amount. An upper bound of  $5 \times 10^{52}$  was given by Victor Allis 1994. – Gunther Piez Dec 3, 2009 at 23:11
- 

Surely with a variable length encoding only the average is at least  $10^{43}$ ? An encoding biased towards more positions must reduce this, particularly as many of the positions are impossible. – Phil H Dec 23, 2009 at 12:02

---

The EveryChess link is now 'for sale', archive.org link:  
[web.archive.org/web/20120303094654/http://...](http://web.archive.org/web/20120303094654/http://...) – oPless Sep 17, 2016 at 2:58

---



4

It'd add interest to optimize for *average-case* size for typical games played by humans, instead of the worst case. (The problem statement doesn't say which; most responses assume worst-case.)



For the move sequence, have a good chess engine generate moves from each position; it'll produce a list of  $k$  possible moves, ordered by its ranking of their quality. People generally pick good moves more often than random moves, so we need to learn a mapping from each position in the list to the probability that people pick a move that 'good'. Using these probabilities (based on a corpus of games from some internet chess database),

encode the moves with [arithmetic coding](#). (The decoder must use the same chess engine and mapping.)

For the starting position, ralu's approach would work. We could refine it with arithmetic coding there as well, if we had some way to weight the choices by probability — e.g. pieces often appear in configurations defending each other, not at random. It's harder to see an easy way to incorporate that knowledge. One idea: fall back on the above move encoding instead, starting from the standard opening position and finding a sequence that ends in the desired board. (You might try A\* search with a heuristic distance equaling the sum of the distances of pieces from their final positions, or something along those lines.) This trades some inefficiency from overspecifying the move sequence vs. efficiency from taking advantage of chess-playing knowledge. (You can claw back some of the inefficiency by eliminating move choices that would lead to a previously-explored position in the A\* search: these can get weight 0 in the arithmetic code.)

It's also kind of hard to estimate how much savings this would buy you in average-case complexity, without gathering some statistics from an actual corpus. But the starting point with all moves equally probable I think would already beat most of the proposals here: the arithmetic coding doesn't need an integer number of bits per move.

Share Improve this answer

[edited Dec 3, 2009 at 10:11](#)

Follow



community wiki

3 revs

Darius Bacon

---

Complexity for storing this info into pool is  $O(n)$ , check my edited answer. – [Luka Rahne](#) Dec 3, 2009 at 7:55

---

ralu, I'm not sure what you're saying, but if you mean your representation of a sequence of moves uses the optimal space in worst case, then I don't contradict that. The idea here is to take advantage of some moves being more likely than others. – [Darius Bacon](#) Dec 3, 2009 at 9:59

---

All you need to find positions which are more likely is using deterministic (and strong) chess engine which sort available move in given position deterministic way. – [Luka Rahne](#) Dec 5, 2009 at 9:24

---



4

Attacking a subproblem of encoding the steps after an initial position has been encoded. The approach is to create a "linked list" of steps.



Each step in the game is encoded as the "old position->new position" pair. You know the initial position in the beginning of the chess game; by traversing the linked list of steps, you can get to the state after X moves.



For encoding each step, you need 64 values to encode the starting position (6 bits for 64 squares on the board - 8x8 squares), and 6 bits for the end position. 16 bits for 1 move of each side.

Amount of space that encoding a given game would take is then proportionate to the number of moves:

$10 \times (\text{number of white moves} + \text{number of black moves})$  bits.

UPDATE: potential complication with promoted pawns. Need to be able to state what the pawn is promoted to - may need special bits (would use gray code for this to save space, as pawn promotion is extremely rare).

UPDATE 2: You don't have to encode the end position's full coordinates. In most cases, the piece that's being moved can move to no more than X places. For example, a pawn can have a maximum of 3 move options at any given point. By realizing that maximum number of moves for each piece type, we can save bits on the encoding of the "destination".

Pawn:

- 2 options for movement (e2e3 or e2e4) + 2 options for taking = 4 options to encode
- 12 options for promotions - 4 promotions (knight, bishop, rook, queen) times 3 squares (because you can take a piece on the last row and promote the pawn at the same time)
- Total of 16 options, 4 bits

Knight: 8 options, 3 bits

Bishop: 4 bits

Rook: 4 bits

King: 3 bits

Queen: 5 bits

So the spatial complexity per move of black or white becomes

6 bits for the initial position + (variable number of bits based upon the type of the thing that's moved).

Share Improve this answer

edited Dec 3, 2009 at 18:08

Follow

community wiki

5 revs

Alex Weinstein

---

Just updated, I meant 128 combinations - clearly less than 128 bits :) :) – [Alex Weinstein](#) Dec 2, 2009 at 8:33

---

- 1 A game state is not the same as a move. Any given position can be thought of a vertex or node, and a legal move can be thought of an directed edge or arrow, forming a (directed acyclic) graph. – [Shaggy Frog](#) Dec 2, 2009 at 8:36
- 

I'm not sure why the negative votes - I'd love to hear people's opinions on the updated idea. – [Alex Weinstein](#) Dec 2, 2009 at 8:42

---

- 1 This doesn't affect your reasoning, but a tiny correction: a pawn can have four moves not including promotions, or 12 moves including promotions. Example pawn at e2: e3, e4, exd3, exf3. Example pawn at e7: e8Q, e8N, e8R, e8B, exd8Q, exd8N, exd8R, exd8B, exf8Q, exf8N, exf8R, exf8B. – [A. Rex](#) Dec 2, 2009 at 20:09
- 

- 1 One minor issue -- 5 bits only encode 32 values. To specify any square on the board you need 6 bits. – [Chris Dodd](#) Dec 2, 2009 at 23:59
- 



I saw this question last night and it intrigued me so I sat in bed thinking up solutions. My final answer is pretty similar



## Basic solution



Assuming a standard chess game and that you don't encode the rules (like White always goes first), then you can save a lot by encoding just the moves each piece makes.

There are 32 pieces total but on each move you know what colour is moving so there's only 16 squares to worry about, which is **4 bits** for which piece moves this turn.

Each piece only has a limited moveset, which you would enumerate in some way.

- Pawn: 4 options, **2 bits** (1 step forward, 2 steps forward, 1 each diagonal)
- Rook: 14 options, **4 bits** (max of 7 in each direction)
- Bishop: 13 options, **4 bits** (if you have 7 in one diagonal, you only have 6 in the other)
- Knight: 8 options, **3 bits**
- Queen: 27 options, **5 bits** (Rook+Bishop)
- King: 9 options, **4 bits** (8 one-step moves, plus the castling option)

For promotion, there are 4 pieces to choose from (Rook, Bishop, Knight, Queen) so on that move we would add **2**

**bits** to specify that. I think all the other rules are covered automatically (e.g. en passant).

## Further optimizations

First, after 8 pieces of one colour have been captured, you could reduce the piece encoding to 3 bits, then 2 bits for 4 pieces and so on.

The main optimization though is to enumerate *only* the possible moves at each point in the game. Assume we store a Pawn's moves as `{00, 01, 10, 11}` for 1 step forward, 2 steps forward, diagonal left and diagonal right respectively. If some moves are not possible we can remove them from the encoding for this turn.

We know the game state at every stage (from following all the moves), so after reading which piece is going to move, we can always determine how many bits we need to read. If we realize a pawn's only moves at this point are capture diagonally right or move forward one, we know to only read 1 bit.

In short, the bit storage listed above for each piece is a *maximum* only. Nearly every move will have fewer options and often fewer bits.

Share Improve this answer

edited Dec 4, 2009 at 13:57

Follow



At each position get the number of all possible moves.

4

next move is generated as



```
index_current_move = n % num_of_moves //this is  
best space efficiency  
n=n/num_of_moves
```



**provably best space efficiency for storing randomly**

**generated game** and need approx 5 bits/move on average since you have 30-40 possible moves.

Assembling storage is just generating n in reverse order.

Storing position is harder to crack, because of great redundancy. (There can be up to 9 queens on board for one site but in that case there are no pawns, and bishops if on the board are on opposite colored squares) but generally is like storing combination of same pieces over remaining squares.)

### EDIT:

Point in saving moves is to store only the index of move. Instead of storing Kc1-c2 and trying to reduce this info we should add only index of move generated from deterministic movegenerator(position)

At each move we add information of size

```
num_of_moves =  
get_number_of_possible_moves(position) ;
```

in pool and this number cannot be reduced

generating information pool is

```
n=n*num_of_moves+ index_current_move
```

## **extra**

If there is only one move available in final position, save as number of previously done forced moves. Example: if starting position has 1 forced moves for each side (2 moves) and we want to save this as one move game, store 1 in pool n.

## **example of storing into info pool**

Lets suppose that we have known starting positions and we do 3 moves.

In first move there are 5 available moves, and we take move index 4. In second move there are 6 available moves and we take position index 3 and in 3th move there are 7 moves available for that side and he chose to pick the move index 2.

Vector form; index=[4,3,2] n\_moves=[5,6,7]

We are encoding this info backwards, so  $n = 4 + 5 \cdot (3 + 6 \cdot (2)) = 79$  (no multiplying by 7 needed)

How to unloop this? First we have position and we find out that there are 5 moves available. So

```
index=79%5=4  
n=79/5=15; //no remainder
```

We take move index 4 and examine position again and from this point we find out that there are 6 possible moves.

```
index=15%6=3  
n=15/6=2
```

And we take move index 3 which gets us to a position with 7 possible moves.

```
index=2%7=2  
n=2/7=0
```

We do last move index 2 and we reach final position.

As you can see the time complexity is  $O(n)$  and space complexity is  $O(n)$ . Edit: time complexity is actually  $O(n^2)$  because the number you multiply by increases, but there should be no problem storing games up to 10,000 moves.

---

## **saving position**

Can be done close to optimum.



When we find out about information and storing informations let me talk more about it. General idea is to decrease redundancy (I will talk about that later). Lets presume that there were no promotions and no taking so there are 8 pawns, 2 rooks, 2 knights, 2 bishops 1 king and 1 queen per side.

What do we have to save: 1. position of each peace 2. possibilities of castling 3. possibilities of en-passant 4. side that has move available

Let's suppose that every piece can stand anywhere but not 2 pieces at same place. Number of ways 8 pawns of same color can be arranged on board is  $C(64/8)$  (binomial) which is 32 bits, then 2 rooks  $2R \rightarrow C(56/2)$ ,  $2B \rightarrow C(54/2)$ ,  $2N \rightarrow C(52/2)$ ,  $1Q \rightarrow C(50/1)$ ,  $1K \rightarrow C(49/1)$  and same for other site but starting with  $8P \rightarrow C(48/8)$  and so on.

Multiplying this together for both sites get us number 4634726695587809641192045982323285670400000 which is approx 142 bits, we have to add 8 for one possible en-passant (en-passant pawn can be in one of 8 places), 16 (4 bits) for castling limitations and one bit for site that has move. We end up with  $142+3+4+1=150\text{bits}$

But now let's go on the hunt for redundancy on the board with 32 pieces and no taking.

1. both black and white pawns are on same column and facing each other. Each pawn is facing other pawn that means that white pawn can be at most at 6th

rank. This brings us  $8 \cdot C(6/2)$  instead of  $C(64/8) \cdot C(48/8)$  which decreases information by 56 bits.

2. possibility of castling is also redundant. If rooks are not on starting place there is no castling possibility with that rook. So we can imaginarily add 4 squares on board to get the extra info if castling with this rook is possible and remove 4 castling bits. So instead of  $C(56/2) \cdot C(40/2) \cdot 16$  we have  $C(58/2) \cdot C(42/2)$  and we lost 3.76 bits (almost all of 4 bits)
3. en-passant: When we store one of 8 en passant possibilities, we know position of black pawn and reduce informational redundancy (if it is white move and has 3th pawn en-passant that means that black pawn is on c5 and white pawn is either c2, c3 or c4) so instead of  $C(6/2)$  we have 3 and we lost 2.3 bits. We decrease some redundancy if we store white en-passant number also side from which can be done (3 possibilities  $\rightarrow$  left, right, both) and we know the position of pawn that can take en passant. (for instance from previous en passant example white black on c5 what can be in left, right or both. if it is on one side we have  $2 \cdot 3$  (3 for storing possibilities and 2 possible moves for black pawn on 7th or 6 rank) instead of  $C(6/2)$  and we reduce by 1.3 bits and if on both sides we reduce by 4.2 bits. That way we can reduce by  $2.3 + 1.3 = 3.6$  bits per en passant.
4. bishops: bishops can be on opposite squares only, this reduces redundancy by 1 bit for each side.

If we sum up we need  $150-56-4-3.6-2=85$ bits for storing chess position if there were no takings

And probably not much more if there are takings and promotions taken in account (but i will write about that later if somebody will find this long post usefull)

Share Improve this answer edited Feb 2, 2010 at 16:58

Follow

community wiki  
12 revs, 2 users 82%  
Luka Rahne

---

Interesting approach. Add some more detail :)

– [Andrew Rollings](#) Dec 2, 2009 at 11:02

---

I added also aproach for saving position. I got down to 85bits on positions whit no taking and is a good ilustration how far is it possible to go. I think that best idea is in saving castling possibilities where almost all off 4 bits are redundant.

– [Luka Rahne](#) Dec 4, 2009 at 11:59

---



3

Most people have been encoding the board state, but regarding the moves themselves.. Here's a bit-encoding description.



Bits per piece:



- **Piece-ID:** Max 4 bits to identify the 16 pieces per side. White/black can be inferred. Have an ordering defined on the pieces. As the number of pieces drops

below the respective powers of two, use fewer bits to describe the remaining pieces.

- **Pawn:** 3 possibilities on the first move, so +2 bits (forward by one or two squares, en passant.) Subsequent moves do not allow moving forward by two, so +1 bit is sufficient. Promotion can be inferred in the decoding process by noting when the pawn has hit the last rank. If the pawn is known to be promoted, the decoder will expect another 2 bits indicating which of the 4 major pieces it has been promoted to.
- **Bishop:** +1 bit for diagonal used, Up to +4 bits for distance along the diagonal (16 possibilities). The decoder can infer the max possible distance that the piece can move along that diagonal, so if it's a shorter diagonal, use less bits.
- **Knight:** 8 possible moves, +3 bits
- **Rook:** +1 bit for horizontal / vertical, +4 bits for distance along the line.
- **King:** 8 possible moves, +3 bits. Indicate castling with an 'impossible' move -- since castling is only possible while the king is on the first rank, encode this move with an instruction to move the king 'backwards' -- i.e. out of the board.
- **Queen:** 8 possible directions, +3bits. Up to +4 more bits for distance along the line / diagonal (less if the diagonal is shorter, as in the bishop's case)

Assuming all pieces are on the board, these are the bits per move: Pawn - 6 bits on first move, 5 subsequently. 7 if promoted. Bishop: 9 bits (max), Knight: 7, Rook: 9, King: 7, Queen: 11 (max).

Share Improve this answer

edited Dec 2, 2009 at 10:46

Follow

community wiki

6 revs

int3

---

32 bits to identify the piece??? I think you meant 5 (32 pieces). Or 6 if you need to encode an 'end' state, – [Toad](#)  
Dec 2, 2009 at 8:51

---

A pawn can also be promoted to rook, knight or bishop. This is common to avoid Stalemate, or avoid confrontation. – [Kobi](#)  
Dec 2, 2009 at 9:02

---

This doesn't affect your reasoning, but a tiny correction: a pawn can have four moves not including promotions, or 12 moves including promotions. Example pawn at e2: e3, e4, exd3, exf3. Example pawn at e7: e8Q, e8N, e8R, e8B, exd8Q, exd8N, exd8R, exd8B, exf8Q, exf8N, exf8R, exf8B. – [A. Rex](#) Dec 2, 2009 at 20:11

---

Maybe I'm misreading, but a pawn can't do en passant on it's first move. Actually you don't need a special "en passant" notation since that's in the game rules - it will just be a diagonal move. The first move is one of 4 options and subsequent moves are up to 3 options. – [DisgruntledGoat](#)  
Dec 4, 2009 at 14:45

---



3

Is the problem to give an encoding that is most efficient for typical chess games, or one that has the shortest worst case encoding?



For the latter, the most efficient way is also the most opaque: create an enumeration of all possible pairs (initial board, legal sequence of moves), which, with the draw-on-thrice-repeated-position and no-more-than-fifty-moves since last-pawn-move-or-capture rules, is recursive. Then the index of a position in this finite sequence gives the shortest worst-case encoding, but also and equally long encoding for typical cases, and is, I expect, very expensive to compute. The longest possible chess game is supposed to be over 5000 moves, with there typically being 20-30 moves available in each position to each player (though fewer when there are few pieces left) - this gives something like 40000 bits needed for this encoding.

The idea of enumeration can be applied to give a more tractable solution, as described in Henk Holterman's suggestion for encoding moves above. My suggestion: not minimal, but shorter than the examples above I've looked at, and reasonable tractable:

1. 64 bits to represent which squares are occupied (occupancy matrix), plus list of which pieces are in each occupied square (can have 3 bits for pawns, and 4 bits for other pieces): this gives 190 bits for start position. Since there can't be more than 32 pieces on board, the encoding of the occupancy matrix is redundant & so something like common

board positions can be encoded, say as 33 set bits plus index of board from common boards list.

2. 1 bit to say who makes first move
3. Code moves per Henk's suggestion: typically 10 bits per pair of white/black move, though some moves will take 0 bits, when a player has no alternative moves.

This suggests 490 bits to code a typical 30-move game, and would be a reasonably efficient representation for typical games.

About encoding the draw-on-thrice-repeated-position and no-more-than-fifty-moves since last-pawn-move-or-capture rules: if you encode the past moves back to the last pawn move or capture, then you have enough information to decide whether these rules apply: no need for the whole game history.

Share Improve this answer

edited Dec 2, 2009 at 11:01

Follow

community wiki

2 revs

Charles Stewart

---

Assume that I'd take a large selection of games and take an average of the results. – [Andrew Rollings](#) Dec 2, 2009 at 11:20

---



3

The position on a board can be defined in 7 bits (0-63 , and 1 value specifying it is not on the board anymore). So for every piece on the board specify where it is located.



32 pieces \* 7 bits = 224 bits



EDIT: as Cadrian pointed out... we also have the 'promoted pawn to queen' case. I suggest we add extra bits at the end to indicate which pawn have been promoted.

So for every pawn which has been promoted we follow the 224 bits with 5 bits which indicate the index of the pawn which has been promoted, and 11111 if it is the end of the list.

So minimal case (no promotions) is 224 bits + 5 (no promotions). For every promoted pawn add 5 bits.

EDIT: As shaggy frog points out, we need yet another bit at the end to indicate whose turn it is ;^)

Share Improve this answer

edited Feb 2, 2010 at 15:30

Follow

community wiki

3 revs, 2 users 88%

Toad

---

and gzip the result afterwards (if the headers won't increase the result) ;^) – Toad Dec 2, 2009 at 8:28

---



Can you improve that by taking into account that some pieces won't ever be found on certain square colours?

– [Andrew Rollings](#) Dec 2, 2009 at 8:33

andrew: actually I can't. I've forgotten to take into account a promoted pawn into a queen (like cadrian's answer suggests). So it looks like I'll actually need another extra bit

– [Toad](#) Dec 2, 2009 at 8:34

I can see how the black and white bishops can be defined together. I wonder about the knights though.. – [int3](#) Dec 2, 2009 at 8:38

1 You're missing non-queen promotions. – [Loren Pechtel](#) Dec 3, 2009 at 4:41



2



I'd use a run length encoding. Some pieces are unique (or exist only twice), so I can omit the length after them.

Like cletus, I need 13 unique states, so I can use a nibble (4 bits) to encode the piece. The initial board would then look like this:



```
White Rook, W. Knight, W. Bishop, W. Queen, W.  
King, W. Bishop, W. Knight, W. Rook,  
W. Pawn, 8,  
Empty, 16, Empty, 16  
B. Pawn, 8,  
B. Rook, B. Knight, B. Bishop, B. Queen, B. King,  
B. Bishop, B. Knight, B. Rook
```

which leaves me with  $8+2+4+2+8$  nibbles = 24 nibbles = 96 bits. I can't encode 16 with a nibble but since "Empty, 0" doesn't make sense, I can treat "0" as "16".

If the board is empty but for a single pawn in the upper left corner, I get "Pawn, 1, Empty, 16, Empty, 16, Empty 16, Empty, 15" = 10 nibbles = 40 bits.

The worst case is when I have an empty square between each piece. But for the encoding of the piece, I just need 13 out of 16 values, so maybe I can use another one to say "Empty1". Then, I need 64 nibbles == 128bits.

For the movements, I need 3 bits for the piece (the color is given by the fact that white always moves first) plus 5 bits (0..63) for the new position = one byte per movement. Most of the time, I don't need the old position since only a single piece will be within range. For the odd case, I must use the single unused code (I just need 7 codes to encode the piece) and then 5 bits for the old and 5 bits for the new position.

This allows me to encode castling in 13 bites (I can move the King towards the Rook which is enough to say what I intend).

[EDIT] If you allow a smart encoder, then I need 0 bits for the initial setup (because it doesn't have to be encoded in any way: It's static) plus one byte per move.

[EDIT2] Which leaves the pawn transformation. If a pawn reaches the last row, I can move it in place to say "transforms" and then add the 3 bits for the piece it is replaced with (you don't have to use a queen; you can replace the pawn with anything but the King).

community wiki

4 revs

Aaron Digulla

---

The smart encoder can't assume that it's an entire game. It could be a fragment of a game. I think you'd still need to encode the start positions. – [Andrew Rollings](#) Dec 2, 2009 at 8:49

---

Well, in the worst case, I need either 128 bits or, if the game is still in the early stage, I can use up to 15 moves to bring it in to the start position = 120 bits. – [Aaron Digulla](#) Dec 2, 2009 at 8:52

---

Since ANY state must be encoded, and not just the initial board state, you also have to encode the pieces itself. So you'll need per piece at least 5 bits. So this will give you at least  $32 \times 5$  bits extra – [Toad](#) Dec 2, 2009 at 8:56

---

@reiner: You're wrong. I just need four bits per piece/empty square. And I already covered this in the first part of my answer, so no " $32 \times 5$  bits extra". For the initial state, I need 96 bits and for any other start state, I need at most 128 bits. – [Aaron Digulla](#) Dec 2, 2009 at 9:00

---

Aaron: still as you say the worst case scenario is really worst case in this encoding. After 3 or 4 moves from a startboard, your encoding will require significantly more bits as you're adding more and more empty's – [Toad](#) Dec 2, 2009 at 9:29

---



Just like they encode games on books and papers: every piece has a symbol; since it's a "legal" game, white

2



moves first - no need to encode white or black separately, just count the number of moves to determine who moved. Also, every move is encoded as (piece,ending position) where 'ending position' is reduced to the least amount of symbols that allows to discern ambiguities (can be zero). Length of game determines number of moves. One can also encode the time in minutes (since last move) at every step.

Encoding of the piece could be done either by assigning a symbol to each (32 total) or by assigning a symbol to the class, and use the ending position to understand which of the piece was moved. For example, a pawn has 6 possible ending positions; but on average only a couple are available for it at every turn. So, statistically, encoding by ending position might be best for this scenario.

Similar encodings are used for spike trains in computational neuroscience (AER).

Drawbacks: you need to replay the entire game to get at the current state and generate a subset, much like traversing a linked list.

Share Improve this answer

answered Dec 2, 2009 at 9:26

Follow

community wiki  
[lorenzog](#)

---

It may only be a game fragment. You can't assume white moves firsts. – [Andrew Rollings](#) Dec 2, 2009 at 9:54

---



2



There are 64 possible board positions, so you need 6 bits per position. There are 32 initial pieces, so we have 192 bits total so far, where every 6 bits indicates the position of the given piece. We can pre-determine the order the pieces appear in, so we don't have to say which is which.

What if a piece is off the board? Well, we can place a piece on the same spot as another piece to indicate that it is off the board, since that would be illegal otherwise. But we also don't know whether the first piece will be on the board or not. So we add 5 bits indicating which piece is the first one (32 possibilities = 5 bits to represent the first piece). Then we can use that spot for subsequent pieces that are off the board. That brings us to 197 bits total. There has to be at least one piece on the board, so that will work.

Then we need one bit for whose turn it is - brings us to **198 bits**.

What about pawn promotion? We can do it a bad way by adding 3 bits per pawn, adding on 42 bits. But then we can notice that most of the time, pawns aren't promoted.

So, for every pawn that is on the board, the bit '0' indicates it is not promoted. If a pawn is not on the board then we don't need a bit at all. Then we can use variable

length bit strings for which promotion he has. The most often it will be a queen, so "10" can mean QUEEN. Then "110" means rook, "1110" means bishop, and "1111" means knight.

The initial state will take  $198 + 16 = \mathbf{214 \text{ bits}}$ , since all 16 pawns are on the board and unpromoted. An end-game with two promoted pawn-queens might take something like  $198 + 4 + 4$ , meaning 4 pawns alive and not promoted and 2 queen pawns, for  $\mathbf{206 \text{ bits}}$  total. Seems pretty robust!

===

Huffman encoding, as others have pointed out, would be the next step. If you observe a few million games, you'll notice each piece is much more likely to be on certain squares. For example, most of the time, the pawns stay in a straight line, or one to the left / one to the right. The king will usually stick around the home base.

Therefore, devise a Huffman encoding scheme for each separate position. Pawns will likely only take on average 3-4 bits instead of 6. The king should take few bits as well.

Also in this scheme, include "taken" as a possible position. This can very robustly handle castling as well - each rook and king will have an extra "original position, moved" state. You can also encode en passant in the pawns this way - "original position, can en passant".

With enough data this approach should yield really good results.

Share Improve this answer

edited Dec 2, 2009 at 9:47

Follow

community wiki

3 revs

Claudiu

- 
- 2 Just assign the removed pieces to the same square as the king. Since the king can never be removed it wouldn't be ambiguous – [John La Rooy](#) Dec 2, 2009 at 10:06
- 

That's a good comment :) Nice aspects to this solution too. I didn't realize it was going to be so hard to pick a winner.

– [Andrew Rollings](#) Dec 2, 2009 at 23:05

---



2



I'd try to use [Huffman encoding](#). The theory behind this is - in every chess game there will be some pieces that will move around a lot, and some that don't move much or get eliminated early. If the starting position has some pieces already removed - all the better. The same goes for squares - some squares get to see all action, while some don't get touched much.

Thus I'd have two Huffman tables - one for pieces, other for squares. They will be generated by looking at the actual game. I could have one large table for every piece-square pair, but I think that would be pretty inefficient

because there aren't many instances of the same piece moving on the same square again.

Every piece would have an assigned ID. Since there are 32 different pieces, I would need just 5 bits for the piece ID. The piece IDs are not changing from game to game. The same goes for square IDs, for which I would need 6 bits.

The Huffman trees would be encoded by writing each node down as they are traversed in inorder (that is, first the node is output, then its children from left to right). For every node there will be one bit specifying whether it's a leaf node or a branch node. If it's a leaf node, it will be followed by the bits giving the ID.

The starting position will simply be given by a series of piece-location pairs. After that there will be one piece-location pair for every move. You can find the end of the starting position descriptor (and the start of the moves descriptor) simply by finding the first piece that is mentioned twice. In case a pawn gets promoted there will be 2 extra bits specifying what it becomes, but the piece ID won't change.

To account for the possibility that a pawn is promoted at the start of the game there will also be a "promotion table" between the Huffman trees and the data. At first there will be 4 bits specifying how many pawns are upgraded. Then for each pawn there will be its Huffman-encoded ID and 2 bits specifying what it has become.



The huffman trees will be generated by taking into account all the data (both the starting position and the moves) and the promotion table. Although normally the promotion table will be empty or have just a few entries.

To sum it up in graphical terms:

```
<Game> := <Pieces huffman tree> <squares huffman tree> <promotion table> <initial position> (<moves> | <1 bit for next move - see Added 2 below>)
```

```
<Pieces huffman tree> := <pieces entry 1> <pieces entry 2> ... <pieces entry N>  
<pieces entry> := "0" | "1" <5 bits with piece ID>
```

```
<squares huffman tree> := <squares entry 1> <squares entry 2> ... <squares entry N>  
<Squares entry> := "0" | "1" <6 bits with square ID>
```

```
<promotion table> := <4 bits with count of promotions> <promotion 1> <promotion 2> ... <promotion N>  
<promotion> := <huffman-encoded piece ID> <2 bits with what it becomes>
```

```
<initial position> := <position entry 1> <position entry 2> ... <position entry N>  
<moves> := <position entry 1> <position entry 2> ... <position entry N>  
<position entry> := <huffman-encoded piece ID> <huffman-encoded square ID> (<2 bits specifying the upgrade - optional>)
```

**Added:** This could still be optimized. Every piece only has a few legal moves. Instead of simply encoding the target square one could give 0-based IDs for the possible

moves of every piece. The same IDs would get reused for every piece, so in total there would be no more than 21 different IDs (the queen can have at most 21 different possible move options). Put this in a Huffman table instead of the fields.

This would however present a difficulty in representing the original state. One might generate a series of moves to put each piece in its place. In this case it would be necessary to somehow mark the end of the initial state and start of the moves.

Alternatively they could be placed by using uncompressed 6-bit square IDs.

Whether this would present an overall decrease in size - I don't know. Probably, but should experiment a bit.

**Added 2:** One more special case. If the game state has NO moves it becomes important to distinguish who moves next. Add one more bit at the end for that. :)

Share Improve this answer

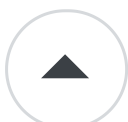
edited Dec 2, 2009 at 9:55

Follow

community wiki

3 revs

Vilx-



[edited after reading the question properly] If you assume every legal position can be reached from the initial

2



position (which is a possible definition of "legal"), then any position can be expressed as the sequence of moves from the start. A snippet of play starting from a non-standard position can be expressed as the sequence of moves needed to reach the start, a switch to turn the camera on, followed by subsequent moves.

So let's call the initial board state the single bit "0".

Moves from any position can be enumerated by numbering the squares and ordering the moves by (start, end), with the conventional 2 square jump indicating castling. There is no need to encode illegal moves, because the board position and the rules are always already known. The flag to turn the camera on could either be expressed as a special in-band move, or more sensibly as an out-of-band move number.

There are 24 opening moves for either side, which can fit in 5 bits each. Subsequent moves might require more or less bits, but the legal moves are always enumerable, so the width of each move can happily grow or expand. I have not calculated, but I imagine 7 bit positions would be rare.

Using this system, a 100 half-move game could be encoded in approximately 500 bits. However, it might be wise to use an opening book. Suppose it contains a million sequences. Let then, an initial 0 indicate a start from the standard board, and a 1 followed by a 20 bit number indicate a start from that opening sequence.

Games with somewhat conventional openings might be shortened by say 20 half-moves, or 100 bits.

This is not the greatest possible compression, but (without the opening book) it would be very easy to implement if you already have a chess model, which the question assumes.

To further compress, you'd want to order the moves according to likelihood rather than in an arbitrary order, and encode the likely sequences in fewer bits (using e.g. Huffman tokens as people have mentioned).

Share Improve this answer

edited Dec 3, 2009 at 11:34

Follow

community wiki

2 revs

Douglas Bagnall

---

Initial position isn't necessarily known. It could be a game fragment. – [Andrew Rollings](#) Dec 3, 2009 at 8:55

---

@Andrew: yes. my mistake. I have edited to allow for game fragments. – [Douglas Bagnall](#) Dec 3, 2009 at 11:37

---



2

If computational time is not an issue then you could use a deterministic possible position generator to assign unique ids to a given position.



From a given position first generate number of possible positions in a deterministic manor, e.g. starting bottom left moving to top right. That determines how many bits you'll need for the next move, some situations it could be as little as one. Then when the move is made store just the unique id for that move.



Promotion and other rules simply count as valid moves as long as they are handled in a deterministic way, e.g. to queen, to rook, to bishop each count as a separate move.

The initial position is hardest and could generate around 250 million possible positions (I think) which would require about 28 bits plus an extra bit to determine whose move it is.

Assuming we know who's turn it is (each turn flips from white to black) the deterministic generator would look something like:

```
for each row
  for each column
    add to list ( get list of possible moves(
current piece, players turn) )
```

'get list of possible moves' would do something like:

```
if current piece is not null
    if current piece color is the same as the
    players turn
        switch( current piece type )
            king - return list of possible king
moves( current piece )
            queen - return list of possible queen
moves( current piece )
            rook - return list of possible rook
moves( current piece )
            etc.
```

If the king is in check then each 'list of possible xxx moves' only returns valid moves that change the check situation.

Share Improve this answer

edited Dec 3, 2009 at 14:39

Follow

community wiki

3 revs

snowdude

---

It's a sneaky solution... so... in this instance, describe your algorithm for generating the deterministic number.

– [Andrew Rollings](#) Dec 2, 2009 at 23:07

---

I did find an interesting link on how long it would take to generate every position on a chessboard :)

[ioannis.virtualcomposer2000.com/math/EveryChess.html](http://ioannis.virtualcomposer2000.com/math/EveryChess.html)

– [Andrew Rollings](#) Dec 3, 2009 at 8:48

---



2



Most of the answers overlooked 3 fold repetition. unfortunately for 3 fold repetition you have to store all the positions played so far...

The question required us to store in space efficient manner so we really don't need to store position as long as we can construct it from the list of moves (provided we have standard starting position). We can optimize PGN and thats it we are done. Bellow is a simple scheme.

There are 64 squares on the board,  $64 = 2^6$ . If we store only the initial and final square of each move that would take 12 bits (Promotion will be tackled later). Note that this scheme already covers player to move, emphassant, piece captured, castling etc; as of these can be constructed from just replaying the move list.

for promotion we can keep a separate array of vectors which would say "at move N promote to Piece XYZ". we can keep a vector of (int, byte).

It is tempting to optimize (To,From) vector also, Since many of these (To,From) vectors are not posible in chess. eg. there wont be a move from e1 to d8 etc. But I couldnt come up with any scheme. Any further ideas are most welcomed.

Share Improve this answer

answered [Dec 23, 2009 at 10:49](#)

Follow



This is how I would encode the game steps. For a game with 40 steps this will take about 180 bits or so.

2



First, create a list of all the choices using an engine that knows all the chess rules. Each step, do this:



1. Enumerate all pieces that are possible to move (at start, white can move 8 pawns and 2 knights, totals 10).
2. Store both the number of possible choices and the choice itself.
3. Enumerate all possible movement positions. (when pawn was chosen at the start, you can move 1 or 2 fields forward, so you have 2 possible choices).
4. Again, store the number of possible choices and the choice itself.

This will get you a list like this:

```
[[10, 3], # choose white pawn at index #3  
 [2, 0], # move it one step forward  
 [10, 2], # choose black pawn #2  
 [2, 1], # move it two steps forward  
 ...  
 ]
```

And so on. To encode it, you just need to store the choice, not the number of possible moves. One way to



store it is to find out how many bits are required for each choice:

```
[[10, 3], # 10 choices => 4 bits
 [2, 0],  # 2 choices => 1 bit
 [10, 2], # 10 choices => 4 bits
 [2, 1],  # 2 choices => 1 bit
 ...
]
```

Totals  $4+1+4+1=10$  bits for the first two moves. But a few bits are wasted, using 4 bits for 10 choices wastes 6 possible choices.

It is possible to do better: reverse the list, and calculate a number based on the possible choices and the choice taken:

```
n = 0          # last position
n = n*2 + 1    # from [2, 1]    n=1
n = n*10 + 2   # from [10, 2]   n=12
n = n*2 + 0    # from [2, 0]    n=24
n = n*10 + 3   # from [10, 3]   n=243
```

Now we have the number **243**, binary **11110011**, which encodes all the above steps in just 8 bits.

To decode, we know that the initial opening position has 10 possible choices. Calculate

```
n = 243
choice = n % 10 # we know there are 10 moveable
               # pieces. => choice=3
n /= 10        # n=24
choice = n % 2  # we know 2 possible moves for
```

```
selected pawn => choice=0
n /= 2          # n=12
choice = n % 10 # 10 moveable pieces for black
player. => choice=2
n /= 10         # n=1
choice = n % 2  # 2 possible moves for pawn =>
choice=1
n /= 2          # n=0, finished decoding
```

Encoding is extremely efficient, especially the endgame because there are not many possible choices left. Also, when you only have one possible move left, you do not need any storage at all for that move.

Share Improve this answer

answered Dec 27, 2009 at 15:31

Follow

community wiki  
[martinus](#)



I've thought about that one for a long time (+- 2hours). And there is no obvious answers.

2

Assuming:



1. Ignoring time state (A player didn't use to have a time limit therefore could force a draw by not playing)
2. When was the game played?!? It's important because the rules have changed over time (so will assume a modern game in the subsequent point a modern game...) Please refer to dead pawn rule for example (wikipedia has a very famous problem



showing it), and if you want to go back in time good luck bishop used to only move slowly and dice used to be used. lol.

... so up to date modern rules it is. First irrespective of repetition and move repetition limit.

-C 25 bytes rounded (  $64b + 32 \times 4b + 5b = 325b$  )

=64 bits (something/nothing) +  $32 \times 4$  bits [ 1bit=color{black/white} + 3bit=type of piece{King,Queen,Bishop,kNight,Rook,Pawn,MovedPawn} NB:Moved pawn... e.g if it was the last moved pawn in the previous turn indicating that an 'en passant' is feasible. ] + 5bit for the actual state (who's turn, en passant, possibility of rooking or not on each sides)

So far so good. Probably can be enhanced but then there would be variable length and promotion to take in consideration!?

Now, following rules are only applicable WHEN a player applies for a draw, IT IS NOT automatic! So consider this 90 moves without a capture or a a pawn move is feasible if no player calls for a draw! Meaning that all moves need to be recorded... and available.

-D repetition of position... e.g. state of board as mentioned above (see C) or not... (see following regarding the FIDE rules) -E That leaves the complex problem of 50 move allowance without capture or pawn move there a counter is necessary... However.

So how do you deal with that?... Well really there is no way. Because neither player may want to draw, or realize that it has occurred. Now in case E a counter might suffice... but here is the trick and even reading the FIDE rules (<http://www.fide.com/component/handbook/?id=124&view=article>) I can't find an answer... what about loss of ability of rooking. Is that a repetition? I think not but then this is a blurred subject not addressed, not clarified.

So here is two rules that are two complex or undefined even to try to encode... Cheers.

So the only way to truly encode a game is to record all from start... which then conflict (or not?) with the "board state" question.

Hope this help... not too much math :-) Just to show that some question are not as easy, too open for interpretation or pre-knowledge to be a correct and efficient. Not one I would consider for interviewing as it open too much of a can of worm.

Share Improve this answer

[edited Nov 3, 2012 at 13:33](#)

Follow

community wiki

[2 revs, 2 users 95%](#)

[sylvain.bouche](#)

---



## Possible improvement on the starting position in Yacoby's solution

2



No legal position has more than 16 pieces of each color.

The number of ways to place up to 16 black and 16 white pieces on 64 squares is about  $3.63e27$ .



$\log_2(3.63e27)=91.55$ . This means you can encode the position and color of all pieces in 92 bits. This is less than the 64 bits for position + up to 32 bits for color that Yacoby's solution requires. You can save 4 bits in the worst case at the expense of considerable complexity in the encoding.



On the other hand, it increases the size for positions with 5 or more pieces missing. These positions represent only <4% of all positions, but they are probably a majority of the cases where you want to record a starting position different than the initial position.

### This leads to the complete solution

1. Encode the position and color of pieces according to the method above. **92 bits**.
2. To specify the type of each piece, use a Huffman Code: pawn: '0', rook: '100', knight: '101', bishop: '110', queen: '1110', king: '1111'. This requires  $(16*1 + 12*3 + 4*4) = 68$  bits for a full set of pieces. The full board position can be encoded in  $92 + 68 =$  **160 bits maximum**.

3. Additional game state should be added: turn: 1 bit, which castling is possible: 4 bits, "en passant" possible: up to 4 bits (1 bit tells it is the case and 3 bits tell which one). The starting position is encoded in  $= 160 + 9 = \mathbf{169 \text{ bits}}$
4. For the list of moves, enumerate all possible moves for a given position and store the position of the move in the list. The list of moves includes all special cases (castling, en passant, and resigning). Use only as many bits as necessary to store the highest position. On average it shouldn't exceed 7 bits per move (16 possible pieces and 8 legal moves per piece on average). In some case, when a move is forced, it requires only 1 bit (move or resign).

Share Improve this answer

edited Aug 24, 2014 at 16:17

Follow

community wiki

2 revs

Florian F



1

There are 32 pieces on the board. Each piece has a position (one in 64 squares). So you just need 32 positive integers.



I know 64 positions hold in 6 bits but I wouldn't do that. I'd keep the last bits for a few flags (dropped piece, queen'ed pawn)





Share Improve this answer

answered Dec 2, 2009 at 8:29

Follow

community wiki  
[cadrian](#)

---

You don't need to use flags to hold state. You can assume that your encoder is smart enough to "know the rules". Thus, if a pawn suddenly changed to a queen, that wouldn't necessarily have to be flagged specifically in the encoding (unless, I suppose, the player chose not to promote).

– [Andrew Rollings](#) Dec 2, 2009 at 8:40

---

yes it should, since you can't tell by the initial position of a pawn if the pawn has been promoted or not! So it as to be encoded in the initial setup – [Toad](#) Dec 2, 2009 at 8:44

---

Ah, but why would you need to know if it's already been promoted? It's just a piece. It's past state would be irrelevant in this instance. – [Andrew Rollings](#) Dec 2, 2009 at 8:46

---

I think if a pawn is still a pawn or has been promoted to a queen is hardly irrelevant for the rest of the game. If you don't think so, I'd love to play a game of chess with you ;^) – [Toad](#) Dec 2, 2009 at 8:59

---

@reinier: He's claiming it's irrelevant if a current **queen** was originally a queen or originally a pawn. – [A. Rex](#) Dec 2, 2009 at 20:05

---



1

**cletus'** answer is good, but he forgot to also encode whose turn it is. It is part of the current state, and is necessary if you're using that state to drive a search algorithm (like an alpha-beta derivative).



I'm not a chess player, but I believe there's also one more corner case: how many moves have been repeated. Once each player makes the same move three times, the game is a draw, no? If so, then you'd need to save that information in the state because after the third repeat, the state is now terminal.

Share Improve this answer

answered [Dec 2, 2009 at 8:34](#)

Follow

community wiki  
[Shaggy Frog](#)

---

going that route, you also need to add the played time for both players since in a real chess game both players can only think 1 or 2 hours total. – [Toad](#) Dec 2, 2009 at 8:39

- 
- 2 You don't have to encode the rules in the actual data. You can assume that the encoder itself knows any rules that are necessary. – [Andrew Rollings](#) Dec 2, 2009 at 8:39

---

Ah.. I didn't consider playing time. Good call... :)  
– [Andrew Rollings](#) Dec 2, 2009 at 8:40

---

@Andrew Rollings: the rule is state-based, as in, it only triggers when a certain precondition is met. Tracking that state of the precondition is also part of the... well, state. :)  
– [Shaggy Frog](#) Dec 2, 2009 at 10:13

---

Irrelevant in this case. If necessary the decoder could examine the state to determine the winner. Remember, the encoder/decoder is rule-aware. The only things that really need to be encoded are the player's choices - anything else can be assumed to be known by the encoder/decoder.  
– [Andrew Rollings](#) Dec 2, 2009 at 10:30

---





1



As several others have mentioned you could for each of the 32 pieces you could store which square they're on and if they're on the board or not, this gives  $32 * (\log_2(64) + 1) = 224$  bits.

However the bishops can only occupy either the black or white squares so for these you only need  $\log_2(32)$  bits for the position, which give  $28 * 7 + 4 * 6 = 220$  bits.

And since the pawns don't start at the back and can only move forward, they can only be on 56, it should be possible to use this limitation to reduce the number of bits needed for the pawns.

Share Improve this answer

edited Dec 2, 2009 at 8:45

Follow

community wiki

2 revs

Andreas Brinck

---

the bishops too can be off the board, so you need an extra bit for those. Also you are forgetting about promoted pawns and the person who is to start first. Taking all this into account you basically end up with my answer ;^) – [Toad](#) Dec 2, 2009 at 8:48

---

The 6 bits for the bishops are  $\log_2(32) + 1 = 6$ , but this sure is a complicated question when you consider all details :) – [Andreas Brinck](#) Dec 2, 2009 at 13:15

---

I was thinking along these lines but it doesn't help. Look at Thomas' answer and modify his Huffman encoding to remove the notion of empty spaces. You use 64 bits to store the matrix of which squares are occupied and you remove 1 bit from each encode--thus exactly recovering the same 64 bits.

– [Loren Pechtel](#) Dec 3, 2009 at 4:57

---



1



A board has 64 squares, and can be represented by 64 bits showing if a square is empty or not. We only need piece information if a square has a piece. Since the player + piece takes 4 bits (as shown earlier) we can get the current state in  $64 + 4 * 32 = 192$  bits. Throw in the current turn and you have 193 bits.



However, we also need to encode the legal moves for each piece. First, we calculate the number of legal moves for each piece, and append that many bits after the piece identifier of a full square. I calculated as follows:

Pawn: Forward, first turn two forward, en passant \* 2, promotion = 7 bits. You can combine the first turn forward and promotion into a single bit since they cannot happen from the same position, so you have 6. Rook: 7 vertical squares, 7 horizontal squares = 14 bits Knight: 8 squares = 8 bits Bishop: 2 diagonals \* 7 = 14 bits Queen: 7 vertical, 7 horizontal, 7 diagonal, 7 diagonal = 28 bits King: 8 surrounding squares

This still means you would need to map the targeted squares based on the current position, but it (should be) a simple calculation.

Since we have 16 pawns, 4 rooks/knights/bishops, and 2 queens/kings, this is  $16 * 6 + 4 * 14 + 4 * 8 + 4 * 14 + 2 * 28 + 2 * 8 = 312$  more bits, bringing the total to 505 bits overall.

As for the number of bits required per piece for possible moves, some optimisations could be added to that and the number of bits probably reduced, I just used easy numbers to work with. For example, for sliding pieces you could store how far away they could move, but this would require extra calculations.

Long story short: Only store extra data (piece, etc) when a square is occupied, and only store the minimum number of bits for each piece to represent its legal moves.

EDIT1: Forgot about castling and pawn promotion to any piece. This could bring the total with explicit positions to 557 moves (3 more bits for pawns, 2 for kings)

[Share](#) [Improve this answer](#)

answered [Dec 2, 2009 at 9:09](#)

[Follow](#)

community wiki  
[Cullen Walsh](#)



Each piece can be represented by 4 bits (pawn to king, 6 types), black/white = 12 values



Each square on the board can be represented by 6 bits (x coord, y coord).



Initial positions require maximum of 320 bits (32 pieces, 4 + 6 bits)



Each subsequent move can be represented by 16 bits (from-position, to-position, piece).

Castling would require an extra 16 bits, as it's a dual move.

Queened pawns could be represented by one of the 4 spare values out of 4 bits.

Without doing the maths in detail, this starts to save space after the first move compared to storing  $32 * 7$  bits (predefined array of pieces) or  $64 * 4$  bits (predefined assignment of squares)

After 10 moves on both sides, the maximum space required is 640 bits

...but then again, if we identify each piece uniquely (5 bits) and add a sixth bit for flagging queened pawns, then we only need piece-id + to-position for each move. This changes the calculation to...

Initial positions = max 384 bits (32 pieces, 6 + 6 bits)

Each move = 12 bits (to-position, piece-id)

Then after 10 moves on each side the maximum space required is 624 bits

community wiki

4 revs

Steve De Caux

---

The second option has the added advantage that storage can be read as 12 bit records, each record = position and piece. The first move can be detected by the fact that the piece has a prior entry. – [Steve De Caux](#) Dec 2, 2009 at 9:34

---

for the time between moves, add x bits for the counter to each record. For the setup records, this will be set to 0. – [Steve De Caux](#) Dec 2, 2009 at 9:43

---

This is the approach I was going to write out. One optimization is that for standard games, you don't need to encode the initial positions at all - a single bit at the head saying "this is a standard game" is sufficient. Also, castling doesn't take a double move - since a castling move is always obvious, and there's only one valid way for the rook to move when a given king half of castling occurs, it's redundant info. For promotion, you can just use the 4 bits after a pawn moves to the last row to specify the new piece type it becomes. – [kyoryu](#) Dec 3, 2009 at 6:58

---

So, for a typical game, after 10 moves you'd be at 121 bits, assuming no promotions. Atypical games would require 1 bit for flag, pieces\*10bits, and another bit to indicate first player. Also, each move would only require 12 bits, as the piece on a given square is implicit from the previous moves in the game. This is probably less efficient than some of the suggested methods, but is pretty clean, and reasonably efficient for "standard" games. – [kyoryu](#) Dec 3, 2009 at 7:00

---

@kyoro - I'm not convinced that 12 bits per move can be beaten - using your idea of a null for standard setup (I would still use 12 bits set to bin zero) - after 1000 moves on each side this is 24012 bits aka 3002 bytes (rounded up) Even using some form of compression you need to cheat to beat this, by declaring your dictionary hard-coded (or logically derivable, same thing) – [Steve De Caux](#) Dec 3, 2009 at 7:27

---



Like Robert G, I'd tend to use PGN since it's standard and can be used by a wide range of tools.

1



If, however, I'm playing a chess AI that's on a distant space probe, and thus every bit is precious, this is what I'd do for the moves. I'll come back to encoding the initial state later.



The moves don't need to record state; the decoder can take keep track of state as well as what moves are legal at any given point. All the moves need to record is which of the various legal alternatives is chosen. Since players alternate, a move doesn't need to record player color. Since a player can only move their own color pieces, the first choice is which piece the player moves (I'll come back to an alternative that uses another choice later). With at most 16 pieces, this requires at most 4 bits. As a player loses pieces, the number of choices decreases. Also, a particular game state may limit the choice of pieces. If a king can't move without placing itself in check, the number of choices is reduced by one. If a king is in check, any piece that can't get the king out of check isn't

a viable choice. Number the pieces in row major order starting at a1 (h1 comes before a2).

Once the piece is specified, it will only have a certain number of legal destinations. The exact number is highly dependent on board layout and game history, but we can figure out certain maximums and expected values. For all but the knight and during castling, a piece can't move through another piece. This will be a big source of move-limits, but it's hard to quantify. A piece can't move off of the board, which will also largely limit the number of destinations.

We encode the destination of most pieces by numbering squares along lines in the following order: W, NW, N, NE (black side is N). A line starts in the square furthest in the given direction that's legal to move to and proceeds towards the. For an unencumbered king, the list of moves is W, E, NW, SE, N, S, NE, SW. For the knight, we start with 2W1N and proceed clockwise; destination 0 is the first valid destination in this order.

- Pawns: An unmoved pawn has 2 choices of destinations, thus requiring 1 bit. If a pawn can capture another, either normally or en passant (which the decoder can determine, since it's keeping track of state), it also has 2 or 3 choices of moves. Other than that, a pawn can only have 1 choice, requiring no bits. When a pawn is in its 7<sup>th</sup> rank, we also tack on the promotion choice. Since pawns are usually

promoted to queens, followed by knights, we encode the choices as:

- queen: 0
  - knight: 10
  - bishop: 110
  - rook: 111
- Bishop: at most 13 destinations when at {d,e}{4,5} for 4 bits.
  - Rook: at most 14 destinations, 4 bits.
  - Knights: at most 8 destinations, 3 bits.
  - Kings: When castling is an option, the king has it's back to S and can't move downwards; this gives a total of 7 destinations. The rest of the time, a king has at most 8 moves, giving a maximum of 3 bits.
  - Queen: Same as choices for bishop or rook, for a total of 27 choices, which is 5 bits

Since the number of choices isn't always a power of two, the above still wastes bits. Suppose the number of choices is  $C$  and the particular choice is numbered  $c$ , and  $n = \text{ceil}(\lg(C))$  (the number of bits required to encode the choice). We make use of these otherwise wasted values by examining the first bit of the next choice. If it's 0, do nothing. If it's 1 and  $c+C < 2^n$ , then add  $C$  to  $c$ . Decoding a number reverses this: if the received  $c \geq C$ , subtract  $C$  and set the first bit for the next number to 1. If  $c < 2^n - C$ , then set the first bit for the next number to 0. If  $2^n - C \leq c < C$ , then do nothing. Call this scheme "saturation".



Another potential type of choice that might shorten the encoding is to choose an opponents piece to capture. This increases the number of choices for the first part of a move, picking a piece, for at most an additional bit (the exact number depends on how many pieces the current player can move). This choice is followed by a choice of capturing piece, which is probably much smaller than the number of moves for any of the given players pieces. A piece can only be attacked by one piece from any cardinal direction plus the knights for a total of at most 10 attacking pieces; this gives a total maximum of 9 bits for a capture move, though I'd expect 7 bits on average. This would be particularly advantageous for captures by the queen, since it will often have quite a few legal destinations.

With saturation, capture-encoding probably doesn't afford an advantage. We could allow for both options, specifying in the initial state which are used. If saturation isn't used, the game encoding could also use unused choice numbers ( $C \leq c < 2^n$ ) to change options during the game. Any time  $C$  is a power of two, we couldn't change options.

Share Improve this answer

answered Dec 2, 2009 at 11:50

Follow

community wiki  
[outis](#)

---



1



Thomas has the right approach for encoding the board. However this should be combined with ralu's approach for storing moves. Make a list of all possible moves, write out the number of bits needed to express this number. Since the decoder is doing the same calculation it knows how many are possible and can know how many bits to read, no length codes are needed.

Thus we get 164 bits for the pieces, 4 bits for castling info (assuming we are storing a fragment of a game, otherwise it can be reconstructed), 3 bits for en passant eligibility info--simply store the column where the move occurred (If en passant isn't possible store a column where it's not possible--such columns must exist) and 1 for who is to move.

Moves will typically take 5 or 6 bits but can vary from 1 to 8.

One additional shortcut--if the encode starts with 12 1 bits (an invalid situation--not even a fragment will have two kings on one side) you abort the decode, wipe the board and set up a new game. The next bit will be a move bit.

Share Improve this answer

answered [Dec 3, 2009 at 5:09](#)

Follow

community wiki

[Loren Pechtel](#)

---



Algorithm should deterministically enumerate all possible destinations at each move. Number of destinations:

1



- 2 bishops, 13 destinations each = 26
- 2 rooks, 14 destinations each = 28
- 2 knights, 8 destinations each = 16
- queen, 27 destinations
- king , 8 destinations



8 pawns could all become queens in worst (enumeration-wise) case, thus making largest number of possible destinations  $9 \times 27 + 26 + 28 + 16 + 8 = 321$ . Thus, all destinations for any move can be enumerated by a 9 bit number.

Maximum number of moves of both parties is 100 (if i'm not wrong, not a chess player). Thus any game could be recorded in 900 bits. Plus initial position each piece can be recorded using 6 bit numbers, which totals to  $32 \times 6 = 192$  bits. Plus one bit for "who moves first" record. Thus, any game can be recorded using  $900 + 192 + 1 = 1093$  bits.

Share Improve this answer

answered Dec 3, 2009 at 8:00

Follow

community wiki  
zufar



## Storing the board state

1



The simplest way I thought of is too first have a array of 8\*8 bits representing the location of each piece (So 1 if there is a chess piece there and 0 if there isn't). As this is a fixed length we don't need any terminator.



Next represent every chess piece in order of its location. Using 4 bits per piece, this takes 32\*4 bits (128 in total). Which is really really wasteful.

Using a binary tree, we can represent a pawn in one byte, a knight and rook and bishop in 3 and a king and queen in 4. As we also need to store the color of the piece, which takes an extra byte it ends up as (forgive me if this is wrong, I have never looked at [Huffman coding](#) in any detail before):

- Pawn : 2
- Rook: 4
- Knight: 4
- Bishop: 4
- King: 5
- Queen: 5

Given the totals:

$$2*16 + 4*4 + 4*4 + 4*4 + 2*5 + 2*5 = 100$$

Which beats using a fixed size set of bits by 28 bits.

So the best method I have found is to store it in a  $8^2 + 100$  bit array

$$8*8 + 100 = 164$$

## Storing Moves

The first thing we need to know with is which piece is moving to where. Given that there are at maximum 32 pieces on the board and we know what each piece is, rather than an integer representing the square, we can have an integer representing the piece offset, which means we only have to fit 32 possible values to represent a piece.

Unfortunately there are various special rules, like castling or overthrowing the king and forming a republic (Terry Pratchett reference), so before we store the piece to move we need a single bit indicating if it is a special move or not.

So for each normal move we have a necessary  $1 + 5 = 6$  bits. (1 bit type, 5 bits for the piece)

After the piece number has been decoded, we then know the type of piece, and each piece should represent its move in the most efficient way. For example (If my chess rules are up to scratch) a pawn has a total of 4 possible

moves (take left, take right, move one forward, move two forward).

So to represent a pawn move we need ' $6 + 2 = 8$ ' bits. (6 bit for initial move header, 2 bits for what move)

Moving for the queen would be more complex, in that it would be best to have a direction (8 possible directions, so 3 bits) and a total of 8 possible squares to move to for each direction (so another 3 bits). So to represent moving a queen it would require  $6 + 3 + 3 = 12$  bits.

The last thing that occurs to me is that we need to store which players turn it is. This should be a single bit (White or black to move next)

## **Resultant Format**

*So the file format would look something like this*

[64 bits] Initial piece locations

[100 bits max] Initial pieces [1 bit] Player turn

[n bits] Moves

*Where a Move is*

[1 bit] Move type (special or normal)

[n bits] Move Detail

*If the Move is a normal move, the Move Detail looks something like*

[5 bits] piece

[n bits] specific piece move (usually in the range of 2 to 6 bits]

*If it is a special move*

It should have an integer type and then any additional information (like if it is castling). I cannot remember the number of special moves, so it may be OK just to indicate that it is a special move (if there is only one)

Share Improve this answer

edited Dec 3, 2009 at 12:00

Follow

community wiki

3 revs

Yacoby

---

1

2

Next