

EFFICIENT SEARCH USING BITBOARD MODELS

Pablo San Segundo, Ramón Galán, Fernando Matía, Diego Rodríguez-Losada, Agustín Jiménez
Intelligent Control Group, Universidad Politécnica de Madrid
pablo.sansegundo@upm.es

Abstract

This paper shows a way to speed up search by using an encoding at bit level to model a particular domain. A bitboard is an unsigned integer whose bits have been given an interpretation in the world of discourse. In models encoded in such a way, states will be described internally as a set of bitboard tuples, whilst operators which allow for transitions between states are essentially bitwise operations over elements belonging to that set.

Given a 64-bit processor word, for each transition it would be theoretically possible to reach another state 64 times faster than with a normal encoding, fast transitions being one of the main issues for efficient search algorithms. We have analysed some other key issues related to efficient bitboard model design and formalised the concepts of well formed heuristics and bitboard guides.

We have used this approach to solve instances of the maximum clique problem thus increasing the performance of one of the fastest algorithms known in the domain for optimal search.

1. Introduction

A bit array (or bitmap in some cases) is an array data structure which stores individual bits in a compact form and is effective at exploiting bit-level parallelism in hardware to perform operations quickly. An example of their application can be found in priority queues of some operative systems (i.e the Linux kernel), where bit at index k is set if and only if the k -th process is in the queue.

The extension of this idea for encoding problems from search domains, so that search can be improved by exploiting this form of parallelism has been, however, surprisingly scarce. One reason for this lies in the lack of expressiveness that can be achieved by bit modeling, which have made researchers look for more abstract forms of knowledge representation. Another, less justified, is an extended *a priori* opinion throughout the scientific community, that the benefits

of parallelism at bit level have a counterpart in the overhead needed to extract information relative to a single bit in the compact bit array. This paper proves otherwise.

Most known examples are taken from board games which is the origin of the term *bitboard*. In chess-playing programs, the bits in a 64-bit bitboard indicate the Boolean value of a particular property concerning the 64 squares of the chessboard [6][7]. One of the first systems to employ bitboards as the basic modeling unit is the KAISSA chess programming team in the Soviet Union during the late 1960s. Since bitboards are the basic units of the first real bit models, we have decided to use this term to refer to this kind of modeling, rather than bit arrays or bitmaps.

As mentioned above, in a typical chess bitboard mode, each bit in a 64-bit bitboard refers to whether some predicate relative to each square on the board holds for a particular chess position. Consider the bitboard whose interpretation is the ‘placement of all white pawns on a chessboard’; a board position can thus be encoded using only 12 of these bitboards (6 for each piece, each piece having two possible colors).

The example shows an obvious advantage when using bitboard modeling: space complexity is reduced by a factor equivalent to the size in bits of the processor word. This fact alone cannot, however, justify the use of modeling at bit level. The key factor is the reduction (possibly exponential) in time complexity resulting from the use of bitwise operations to traverse states or to encode procedural knowledge during search.

Amongst the disadvantages of this approach lie the potential overhead when referencing a particular bit of a bitboard, and the difficulty in defining efficient complex operators when working at bit level, a procedure which some researchers describe as an art. Some work on this topic can be found in [13] and [11]. Thus, key issues to be answered are:

1. Which domains are subject to bitboard encoding?
2. What are the key elements in bitboard models that can improve search performance compared to non bitboard models?
3. How can additional knowledge be added to a search algorithm in an efficient way?

These questions are dealt with in the next two sections. The final part of the paper is devoted to the solving of the maximum clique problem (MCP), a classical combinatorial problem known to be NP-hard [4], taken from the graph domain.

The intuition behind the maximum clique problem can be formulated as follows: given an undirected graph G , find the maximal set of vertices in G such that all of them are adjacent to each other. We have built a bitboard model and adapted one of the fastest known algorithms for solving sparse graphs to guide search [8]. As a result the overall performance has been increased, in some instances impressively.

2. Bitboard models

For the sake of simplicity and to be able to establish a comparison in complexity between bitboard and non bitboard models, we will only consider in this paper problem domains that can be fully described in propositional logic in an efficient way. In [11] we discussed a formal method to encode more complex problems, thus showing that the techniques we will describe in this and the following sections could also be extended to other domains, with a possible overhead in time complexity.

To model a domain in propositional calculus, propositional predicates have to be given a correct *interpretation* for that domain; that is, a possible state of the world is a mapping of each propositional symbol to the binary set {true, false}, an assertion about their truth value in that domain.

A binary encoding of the interpreted set of all propositions is a mapping of each symbol to a bit in the set of bitboard tuples which model the world. The actual truth value of the symbol at a given moment is its mapped bit state, either 0 or 1. The truth value for a predicate sentence can be obtained by appropriate bitwise operations at bit level.

Let w_{size} be the size of the processor word (usually 32 or 64 bits for commercial CPU's, although most compilers will allow for 64 bit number declarations). An obvious advantage of this encoding is space reduction by a factor w_{size} , this being the maximum number of bits in a bitboard. Let B be a set of bitboard tuples of size n ; then a set of k elements in B can encode up to $w_{size} \cdot n \cdot k$ propositional symbols about a given domain.

2.1. Bit operators

Reasoning in a bitboard model must be implemented at bit level, the primitive operators being AND, OR and NOT of Boolean algebra, available in most high level languages. In C, for example, $\&$, $|$, \wedge and \sim

correspond with the AND, OR, XOR and NOT at bit level, when applied to two unsigned numbers.

The main advantage of bit logic is that a CPU can do logical operations on two bitboards of w_{size} at the same time, thus simultaneously reasoning with w_{size} propositional symbols over the domain of discourse. This means that a chunk of procedural knowledge can theoretically be processed w_{size} times faster than in a non bitboard model.

As states are fundamentally sets of bitboard tuples, higher level operators related to those from set theory are needed. Here is a list of some of them written in C style, where B and A are sets of bits and $|A| \leq w_{size}$, $|B| \leq w_{size}$ (we assume that bits set to one indicate membership to that particular set).

- $A \cap B : A \& B$
- $A \cup B : A | B$
- $\overline{A} : \sim A$
- $(A \cup B) - (A \cap B) : A \wedge B$
- $A - (A \cup B) : (A \wedge B) \& A$
- $B - (A \cup B) : (A \wedge B) \& B$
- $A - B : A \& (\sim B)$
- $A \supseteq B : B \& (\sim A) \text{ is } 0$

The last operator is not an assignment over sets A and B , but a truth assertion. Thus, the statement $A \supseteq B$ holds for a particular state of the world iff the expression $B \& (\sim A)$ gives zero as a result. Proof is trivial, as $B \& (\sim A)$ set to zero those bits in B which have a value of 1 in A . If there is a non-zero result this means that the set which contains those non-zero bits is not in A , but it clearly belongs to B , so $A \supseteq B$ cannot hold.

2.2. Complexity

When comparing bitboard and non bitboard models, upper bounds in space and time complexity improvement are a function of the CPU word size (w_{size}), the former being constant (a compression factor), the latter being linear on the number of state transitions t during search.

Consider a possible world D made up of n entities with every entity having k different predicates, potentially true or false in a concrete state of the world. Under these hypothesis, and given $n < w_{size}$, only k bitboards are needed to encode the domain, each bitboard having mapped n bits out of a possible w_{size} . Reasoning over a particular element of the domain requires operators such as *most significant bit*, *least significant bit*, or *bit population count* (better known also as MSB, LSB or PC). Efficient operators for 32 or 64 bit numbers can be found in [13] and [11].

However, if n is greater than the word size, k bitboard tuples will be needed, each tuple holding n/w_{size} bitboards. Extensions for the above mentioned operators for bitboards holding more than w_{size} bits introduce an overhead (loop control etc.), a side effect which is by no means irrelevant (as found from our practical experience). This suggests a possible decomposition of the problem space: *whenever possible, partitions of the bitboard model where $|D| < w_{size}$ should be preferred above all other criteria.*

A comparison between search over bitboard and non bitboard models considering a domain D with $|D|=n$, k predicates, $n > w_{size}$ and that each transition involves every k predicate of a particular state:

- Upper bound on SPACE complexity improvement: w_{size} compression factor.
- TIME complexity for a propositional logic non-bitboard model: $k \cdot n \cdot T(n)$
- TIME complexity for a bitboard model: $(k \cdot n / w_{size}) \cdot T(n / w_{size})$

where T is the number of transitions (the same order of magnitude as the number of nodes examined), which we have assumed to be a function of the number of entities of the domain, a quite common situation.

Since T could well be exponential in n/w_{size} this shows that it would be theoretically possible to have an exponential reduction in time complexity just because of an encoding! This does not mean that if a search domain is in NP we are expecting this to change, but the overall complexity of a search procedure could possibly be reduced even exponentially by a factor related to w_{size} in time.

3. Search

In this section we dwell on several specific design issues relevant to bitboard modeling of a general search problem. We start by formalizing a general search problem and a correct bit encoding. Next we make some general suggestions as to the key elements involved in decisions during design and finally we discuss heuristic design and introduce the concepts of well formed heuristics and bitboard guides mainly based on our practical experience over bitboard design.

A SEARCH MODEL

Let D be a search domain which can be modeled in propositional calculus. Let S be the set which contains all possible states of the model, that is, the complete space to be searched. Let G be a goal state for that domain. Under these hypothesis, a general search problem can be formulated by the pair $\langle S_0, G \rangle$, where $S_0 \in S$, which consists in reaching the goal state G from the initial state S_0 by way of traversing other

states in S according to some specified rules usually defined as transition operators.

We assume that our search domain has n entities subject to description, that is $|D|=n$. We will also assume that k predicates in propositional calculus are enough to describe each and every entity in D . Let R be a mapping relation which associates a truth value of some predicate about any entity in D with a bit to 1 of an unsigned number called bitboard. Then, under these assumptions, a *correct encoding* of D is such that for every state in S there exists a mapping through R to a particular value of a subset of all possible bitboards used in the encoding.

More formally, let B be a set of bitboard tuples where $|B|=k$, k being the the number of predicates. A bitboard encoding of D is *correct* if

$$\forall s \in S, \exists R / s \xrightarrow{R} B = \{bb_1, bb_2, \dots, bb_k\} \quad (1)$$

Proof is trivial if we consider that every bb_i encodes all possible truth values of the i -th predicate in D .

To really exploit the bitboard model in a search problem, here are some relevant design issues:

- *Transition operators* should be some simple logical expression applied at bit level, implemented in an efficient way. This principle can be extended to additional procedural knowledge.
- Predicate bitboard tuples (bb_i) should be chosen so *that maximum meaningful information can be extracted from every CPU bitwise operation.* As noted, for every CPU bit operation there is a potential benefit of w_{size} .
- *Individual bit reasoning should be minimized.* When the actual position of a 1-bit relative to other bits in a bitboard needs to be known, there is an overhead; this overhead increases by an order of magnitude in the more general case where domain entities are bigger than w_{size} .
- *Procedural bitboard encoded knowledge can efficiently traverse a search graph.* Bitboards are compact data structures, and therefore can encode knowledge which traverses a search graph with much less penalty in TIME or SPACE than equivalent more conventional structures.

This final point is crucial and should condition the election of semantics behind bb_i state descriptors. A good initial choice may allow defining efficient heuristics for that domain at a later stage. We deal with heuristics for bitboard models in the following subsection.

3.1. Heuristics for bitboard models

In most search domains it is possible to point out key state descriptors which are particularly relevant during search. For example, goal states may have common descriptors; a set of leaf nodes may have a relevant description in a tree search etc.

This kind of procedural knowledge is a key factor for the global performance of the search algorithm and influences model design as well as implementation. Because of the facility of bitboard data structures to traverse a search graph noted in the previous subsection, designers would want to find a heuristic made up of a small set of bitboards that would ‘guide’ search. We formalize these ideas introducing the terms of *well formed heuristics* and *bitboard guides* related to bitboard models in this subsection.

For our search problem $\langle S_0, G \rangle$ we assume it’s possible to define a metric M in S such that for every state in S we can evaluate a distance to G . More formally, for every state S_i in S , there exists a distance function d which maps S_i to the real number set \mathfrak{R} through M .

Let h be the same generic distance function implemented in a bitboard model. In this case h , in its most general form, is a mapping $h: B \xrightarrow{\text{distance}} \mathfrak{R}$ where B is the set of all bitboard tuples which make up the state encoding. We now describe several possible improvements for h over the general case, which should be considered during bitboard model and heuristic design. These ideas have been obtained from our own practical experience.

A first improvement of h is to consider a mapping over the natural numbers, $h: B \xrightarrow{\text{distance}} \mathfrak{N}$, made up of efficient bitwise operators. Out of all the possible metrics in a bitboard space, a well known one is the *Hamming distance* [5]: the number of different bits that have changed between two bit arrays. This can be extended easily to define a metric for states in S .

A particular instance of the above case is a mapping of h with the Boolean truth values 1, 0; that is $h: B \xrightarrow{\text{distance}} \{0, 1\}$. A simple way to define a heuristic of this type is the matching of the bitboard tuples that make up the state with any subset of B (i.e. the goal state G). A 1 result would indicate a perfect match, 0 would indicate otherwise.

A final important improvement would be to define a valid heuristic for our domain D which mapped a reduced number of bitboard tuples (compared to the number of predicates k) with \mathfrak{R} (or even better, \mathfrak{N}). Note that this reduced set does not even have to be in B , meaning that it need not belong to the predicates which formed part of the initial state description in the first place, but make part of additional procedural knowledge worked out during the search process.

All these heuristic design ideas are not always possible but should condition design. To evaluate heuristics for bitboard search models, from an

efficiency point of view, we now introduce two new concepts: a *well formed* heuristic and a *bitboard guide*.

‘WELL FORMED’ HEURISTICS

Let D be a search domain which can be described by k propositional calculus predicates and which has been encoded at bit level. We then say that a heuristic h is *well formed* for D and we denote it by h_{wf} when h is a valid heuristic for that domain and

$$h_{wf}: C \xrightarrow{\text{distance}} \mathfrak{N} / |C| \ll k \quad (2)$$

where C is a set of bitboard tuples if $|D| > w_{size}$ or a set of bitboards if $|D| < w_{size}$. Note that it is not necessary for C to belong to the initial state encoding, but could be additional procedural knowledge evaluated along the search for every state. Throughout search, knowledge encoded in C traverses from node to node and could be the basis for decisions such as cuts or node expansion, acting as a guide. The final part of this subsection formalizes this idea.

BITBOARD GUIDES

Let D be the search domain used in (2). We say a *well formed* heuristic in that domain as defined in (2) is a *bitboard guide* for D if $|C| = 1$. As in the previous case, C is a bitboard tuple if $|D| > w_{size}$, and a simple bitboard if $|D| < w_{size}$.

The importance of bitboard guides is that they implement efficiently a semantic entity which will be repeatedly used during the whole search procedure. To find these critical concepts and, if possible, define bitboard guides for each of them is fundamental during model design.

Consider the vertex coloring problem taken from the graph domain, where the aim is to color all vertices in a graph with the restriction that vertices adjacent to each other must be of different colors. A bitboard guide to reason with leaf nodes during tree search could be the bitboard tuple whose bits express the fact that vertices from the domain are colored or remain uncolored. Note that this chunk of knowledge is not necessarily obvious from the problem description (we could consider, for example, predicates which only take into account the colors of each of the vertices, but not if they are colored or uncolored). In the next section we use an example taken from the clique domain as a case study.

4. The Maximum Clique Problem (MCP)

To test the viability of bitboard modeling we have chosen a classical combinatorial problem from the

graph domain, the maximum clique problem (MCP). MCP has been deeply studied and is known to be NP-hard [4].

After building a bitboard model we have adapted one of the fastest known optimal search algorithms in the domain [8], based on [3], and compared it with the original algorithm for a number of instances. Computational results obtained prove the validity of our approach and open up a new line of research in the field.

4.1. Introduction

Given an undirected graph G made up of vertices V and edges E , $G = \{V, E\}$, two vertices are said to be adjacent if they are connected by an edge. A *clique* is a subgraph of G where any two of its vertices are adjacent.

Typical problems from the clique domain are the *k-clique* problem which is to find a clique of k vertices, and the *maximum clique* which looks for the maximum possible size of a clique for a given graph. Other important graph problems are the *independent set problem* (where the aim is to find subgraphs with every pair of vertices not adjacent) or the *vertex cover problem*. All of them are known to be NP-hard [4].

Clique study has several important practical applications such as computer vision, signal processing and design of microelectronic. It can also be used for backward reasoning in rule based models. Approaches to the problem include stochastic methods without proving optimality and optimal search. The latter direction is the one chosen as our case study.

Since the 70s, many papers have been published with algorithms for the maximum clique problem. To compare these algorithms, DIMACS¹ has set a benchmark of graph instances available electronically. In this paper we have used these and some random graph instances to test our bitboard model.

4.2. Existing algorithms

The main paradigms for optimal clique algorithms are *backtracking* and *branch and bound*. Early work includes [2] to find all the cliques of a graph and [12] for the maximum independent set problem. More recent algorithms for the maximum clique problem, which use some form of branch and bound are [3], [9] and [10], among others.

The basic form of most of the recent published MCP algorithms is that of [3]. Figure 1 illustrates this algorithm in its most general form: $N_{adj}(v_i)$ returns the set of adjacent vertices to v_i ; max_size is a global

variable which stores the best clique found so far and $N_{sel}(U)$ is any node selection strategy to chose a vertex from the set of vertices U .

```

Initialization: U=V, size=0
function clique(U,size)
Step 1:Leaf node (|U|=0)
    1. if (max_size>size) record max_size
    2. return
Step 2:Fail upper bound (size +|U|<max_size)
    1. return
Step 3:Expand node from U
    1.  $v_i:=N_{sel}(U)$ 
    2.  $U:=U \setminus \{v_i\}$ 
    3. clique( $U \cap N_{adj}(v_i)$ ,size+1)

```

Figure 1: basic form of MCP

The algorithm shown in figure 1 implements simple backtracking through recursion. Step 2 implements the branch and bound strategy, node expansion failing when it is impossible to find a better clique than in previous branches. As an upper bound for a maximum clique the cardinality of U is used. This upper bound can be improved by various forms of vertex coloring.

Another interesting idea which seems to work well for a large number of instances is to number the vertices $V=\{v_1, v_2, \dots, v_n\}$ and solve iteratively the maximum clique problem for $V=\{v_n\}$, $V=\{v_{n-1}, v_n\}$, \dots , $V=\{v_2, v_3, \dots, v_n\}$, $V=\{v_1, v_2, \dots, v_n\}$, the last instance being the original problem to be solved. Somewhat surprisingly, if in each iteration max_clique is stored, it can be used as an even better upper bound (and faster to compute) than vertex coloring for many instances. This turns out to be one of the best known algorithms for optimal search so far [8] and it is the one that we have decided to adapt to guide search in our bitboard model.

4.3. Bitboard modeling

To build a bitboard model for an undirected graph G there is only to make a direct mapping between edges and bits. Furthermore, a bitboard tuple bb_i is mapped directly to a vertex v_i of G , each 1-bit representing an edge with the corresponding vertex according to the position of the bit in the tuple. An example: for a graph G of 5 vertices, $bb_3 = \{1, 0, 0, 1, 1\}$ encodes the fact that the third vertex in G is adjacent to vertices 1, 4 and 5.

To guide search we use a further bitboard tuple which points to every adjacent node to nodes already expanded along the search path, which is precisely set U in the algorithm in figure 1. Following our definition

¹ DIMACS: (Center for Discrete Mathematics and Theoretical Computer Science)

in section 3, a well-formed heuristic for leaf nodes would map this new bitboard tuple with the null tuple (a tuple with all bits to zero).

We have then adapted [8] as our search algorithm. It is important to note that to achieve a high overall performance some previous important work is needed to implement basic operators at bit level; such as finding the relative position of a particular 1-bit in relation to others in a bitboard tuple. Work done in this field is mentioned in [11] and is out of the scope of this article.

4.4. Vertex ordering

For the algorithm to work properly, a node strategy is needed to choose the order of the vertices to expand. It is also thought that a good initial ordering based on coloring can also improve search. For our new bitboard model we have used our own personal initial ordering greedy heuristic, which has a [1] flavor but numbers nodes from 1 to n:

When constructing a new color class, first we build the graph induced by the uncolored vertices. Then, as long as there is a vertex yet to be taken, we add to the color class the one which has minimum degree (number of incident edges). The vertices are numbered in the same order they are added to the color class starting from 1 ($v_1, v_2, v_3 \dots$).

Node selection during search simply respects the initial bitboard ordering.

5. Experimental results

Size	Density	Non BB ²	BB
100	.6	<1	1
100	.7	<1	1
100	.8	<1	1
100	.9	3	1
100	.95	<1	< 1
200	.4	<1	< 1
200	.5	<1	1
200	.6	<1	1
200	.7	14	7
200	.8	660	159
300	.6	16	12
300	.7	542	254
500	.5	32.5	30
500	.6	821	667
1000	.4	104	150

Table 1: Random graphs

To evaluate our model we have carried out some tests and compared our results with a similar algorithm working over a non bitboard model. Computational results are shown in tables 1 and 2.

Entries for the existing algorithm in both tables have been obtained directly from [8] (unfortunately the source code has not been made available by the author). In those tests, a 500 MHz Linux PC was used, and entries come under the *Non BB* header.

We first examine some instances of random graphs (see table 1). Entries for our new algorithm come under the *BB* header, showing average run times in CPU seconds on a 2 GHz PC running on a Windows operating system. For each instance, results are displayed which average 5 different runs. No time adjustment has been done to take into account the difference in machines and times have been rounded up to seconds.

DIMACS	n	d	Non BB ¹	BB
brock200_1	200	.75	54	30
c-fat200-1	200	.08	<1	< 1
c-fat200-2	200	.16	<1	1
c-fat200-5	200	.43	<1	< 1
c-fat500-1	500	.04	<1	1
c-fat500-2	500	0.07	<1	1
c-fat500-5	500	.19	10443	1
c-fat500-10	500	.37	<1	1
hamming6-2	64	.09	<1	< 1
hamming6-4	64	.35	<1	< 1
hamming8-2	256	.97	<1	2
hamming10-2	1024	.99	2.5	> 10000
johnson8-2-4	28	.56	<1	< 1
johnson8-4-4	70	.77	<1	< 1
johnson16-2-4	120	.76	<1	1
keller4	171	.65	<1	1
Mann_a9	45	.93	<1	< 1
Mann_a27	378	.99	>10000	13857
san200_0.9_1	200	0.9	< 1	1
san200_0.9_2	200	0.9	4	2

Table 2: DIMACS benchmark

Results indicate that our bitboard model outperforms the previous algorithm when instances get harder. This tendency is more acute for the hardest instances, such as a 200 node graph with 0.8 density or a 300 node graph with 0.7 density. For easy instances (CPU times not more than 1 second) the overall performance of the old algorithm is slightly better. This could be explained by the extra computational overhead needed to manage bitboard structures. Bitboards only show their worth in the long run.

A similar analysis can be made from the tests carried out on DIMACS instances, available electronically in [<URL:ftp://dimacs.rutgers.edu/pub/>](http://ftp://dimacs.rutgers.edu/pub/)

² PC: 500 MHz, Linux

challenge/graph/benchmarks/cliq>. The trend is only broken by the instance *hamming10-2* where the bitboard model is clearly outperformed. This difference may well reside in the fact that node ordering strategies used in our case are not identical to the ones used in [8]. This factor can be of exponential time importance, especially more so in specifically generated hard instances as is the case in the DIMACS benchmark. A similar situation occurs with *c-fat500-5*, but this time tables are turned in favor of the bitboard model.

6. Conclusions

We have showed a general way to improve the performance of search models by building a bitboard model and searching through the bitboard space. Bitboard models give an interpretation to bits and every logical operation on a bitboard is simultaneously operating on w_{size} symbols of the domain.

However, bit operations have a potential source of overhead when reasoning with specific bits, so building an adequate model becomes critical for a good overall performance. We have also dealt, in this paper, with several design issues, formalizing concepts such as a *correct* bitboard encoding, a *well formed* heuristic and a *bitboard guide*. Attention must be paid during design so that, whenever possible, well defined heuristics can be used on single bitboard tuples to guide search. The computation of the heuristics should be implemented by efficient bitwise operators.

Finally, we have tested these techniques to improve the overall efficiency of one of the fastest known optimal algorithms for the maximum clique problem. Results are near the best (if not the best) obtained so far for some instances.

7. Future Work

The results shown in this paper are so encouraging that they might open up a whole new line of research. Work on formalization of the key elements of bitboard models and their impact on search, as well as new tests on concrete problems is badly needed to further evaluate the potential of our approach. For this purpose, combinatorial problems like the maximum clique problem presented in this paper seem to be well suited.

8. References

- [1] N. Biggs, *Some Heuristics for Graph Coloring*, R. Nelson, R.J. Wilson, (Eds.), Graph Colourings, Longman pages: 87-96, New York, 1990.
- [2] C. Bron, J. Kerbosch. *Finding all cliques of an undirected graph*, Comm. ACM 16(9), pages: 575-577, 1973.
- [3] R. Carraghan, P.M. Pardalos. *An exact algorithm for the maximum clique problem*, Oper. Res. Lett. 9: pages 375-382, 1990.
- [4] M.R. Garey, D.S. Johnson. *Computers and Intractability: A guide to the theory of NP-completeness*. Freeman, New York, 1979.
- [5] R. W. Hamming. *Error-detecting and error-correcting codes*, Bell System Technical Journal 29(2):147-160, 1950.
- [6] E.A. Heinz. How DarkThought plays chess, *ICCA Journal*, 20(3): pages 166-176, 1997
- [7] E.A. Heinz, *Scalable Search in Computer Chess*. Vieweg, 2000.
- [8] *A fast algorithm for the maximum clique problem*. Discrete Applied Mathematics, 120(1--3):pages 197--207, 2002
- [9] P.M. Pardalos, J. Xue, *The maximum clique problem*. Global Optimization. 4: pages 301-328, 1994.
- [10] P.M. Pardalos, G.P. Rodgers, *A branch and bound algorithm for the maximum clique problem*, Comput. Oper. Res. 19(5): pages 363-375, 1992.
- [11] Pablo San Segundo and Ramón Galán. *Bitboards: A new approach*. Artificial Intelligence and applications (AIA) Innsbruck, Austria 2005.
- [12] R.E. Tarjan, A.E. Trojanowski. *Finding a maximum independent set*, SIAM. J. Comput. 6(3), pages 537-546, 1977.
- [13] H.S. Warren Jr. *Hacker's Delight*. Addison-Welsey 2002.