findmagic: Finding library subroutines in stripped statically-linked binaries

Katharina Bogad, Technische Universität München

Findmagic is an algorithm designed to recover symbols in statically linked binaries using an attributed graph matching technique. It is designed to be easily adoptable to any given architecture or ABI.

Introduction

In modern reverse engineering, the recovery of standard library functions still poses a problem to the reverse engineering community. Usually, statically linked binaries, especially from commercial sources, ship without symbols. In non-trivial scenarios, i.e. on embedded devices or uncommon architectures, automatic symbol recovery is often impossible due to the lack of suitable reference files. Although numerous methods have been described to compare different versions of the same executable, notably [?] and [?], they focus on carrying reverse engineering work from an earlier binary to a patched version for vulnerability analysis. Starting from the beginning, correctly identifying standard library functions mostly relies on *pattern matching* based algorithms like [?]. In this paper, I present a novel of approach using attributed graphs to identify library functions. The paper is organised as follows: first, a general problem description explaining why and when the pattern matching approach works great - and when it does not. Second, an in-depth description of the proposed algorithm, followed by third, the experimental results, ending in fourth, final notes and conclusions.

Problem description

The most popular debuggers available, IDA Pro and radare2, both employ ways to recover symbols in a stripped statically-linked binary. In IDA Pro, the recognition is based on a pattern-matching technique [? , P. 211 ff] using signatures generated from the object files that were used to link the binaries. However, this relies heavily on the assumption that the library used to link against is available for signature generation as the output of the library compilation process may vary drastically depending on the compiler flags used to compile. Especially the chosen optimization level changes for example how and when gcc inlines functions [?], which in turn creates different binary patterns.

In an environment where the producer of the analyzed binary does not have control over the source code or at least cannot compile a version of the library himself, these pattern matching techniques work great, as there exist only a relatively small number of different versions for this library. Even for runtime libraries like MSVCRT, the Visual C Runtime from Microsoft, which is closed-source, where different versions with different patch-levels exist, a pattern matching technique is entirely feasible since the reverse engineer has all versions that could have possibly been used to link against at hand – either on the harddrive or downloadable from the web. This applies to all other closed-source libraries as well.

However, in an open-source environment the supplier of the target binary could choose to compile the library himself, offering control of the optimization level. If he chooses not to inline any functions (i.e. for debugging purposes), reference patterns from an object file with optimizations enabled would most likely not match. This is already a problem considering binaries that were compiled on different Linux distributions as most of them use their own buildfarm to build their packages. Thus, there are differences in the generated object code as different compiler flags and different compiler versions are used. As long as one of these "standard" libraries is used, one could however guess the exact library used for linking, download it and generate the definition file and start matching.

Leaving the standard Linux distributions and examining binaries for embedded devices, such as wireless lan routers, the libraries used are often optimized to a small size, containing only a subset of functions of the comparable standard library. Projects like dietlibc[1] or uClibc[2], that do not directly provide up to date prebuilt packages and build environments further enhance the difficulty to find suitable object files. Most likely, there is no way of finding out which library exactly was used for linking. Even if the exact library and version were known, the compiler version is still important for branch prediction, which in turn is important for how the blocks of the non-linear code flow in a function are ordered in the binary. In practice, there is no reliable way to guess any of those.

[?] is an algorithm specifically designed for standard C libraries such as the GNU C Library (`glibc`). The algorithm searchs for system calls, for example `int 80h`, and looks up their number in the corresponding system-call table. Using flexible pattern matching which allows multiple matches, wrapper functions for these system calls are identified. This method has multiple drawbacks; the most significant being that it cannot be applied to library functions that do not use system calls such as `strcpy`, which is very interesting in vulnerability analysis.

[?], although focusing on preserving reverse-engineering work between library versions, is similar to the approach presented in this paper. Flake defines an executable as a *graph of graphs*, using the complete control flow of a function, including its calls, as the semantic equality condition.

[1]http://www.fefe.de/dietlibc/
[2]http://uclibc.org/

Algorithm Design

Generally, a program is defined as a set of attributed graphs $G = (N, B)$ with the nodes N as the functions and branches B as the calls between them. A *constant* is defined as an immediate operand to one of the instructions `and`, `or`, `xor` or `mov`. Note however that immediates passed to the `mov` instruction (or its variants) that are adresses to strings outside the code section are not considered a constant. A *string* is defined as an ordered set str of tuples $s = (i, c)$, with i being the index in the string an c the character number, fulfilling these conditions:

$$\begin{aligned}&(\forall (i,c) \in str : c \geq \texttt{0x20} \wedge c \leq \texttt{0xDF}\\ &\quad \vee c = \texttt{0x0A} \vee c = \texttt{0x0D} \vee c = \texttt{0x09} \vee c = \texttt{0x00})\\ &\quad \wedge |str| > 1\\ &\quad \wedge (\forall (i,c) \in str | i = \max(i, str) : c = \texttt{0x00})\\ &\quad \wedge (\forall (i,c) \in str | i \neq \max(i, str) : c \neq \texttt{0x00})\end{aligned} \tag{1}$$

Each node can be defined as a set $N = (n, s, C, S, I)$ with n being the function name, s the address of the function in the binary, C a multiset of the constant values used in this function, S a multiset of the cross-referenced strings used in the function and I an ordered multiset of the machine instructions of the function.

The general purpose of the algorithm is to generate a bijective mapping $M = N_1 \rightarrow N_2$ between a known library function N_1 and a function N_2 inside a statically linked binary. The generation of this mapping is done in multiple steps. A high-level overview of these steps is given below; they will be explained in detail thereafter.

1. Acquire an instance/object file of the target library that includes debug symbols
2. Analyse the object file and build the attributed function graphs
3. Analyse the target binary, build its attributed function graphs
4. Match the graphs of the input object file and the analysed binary

Object file acquisition

In theory, the algorithm is ineffective when it comes to function inlining, as this could add constants or strings to the function that were not present in the input object file and vice versa. In practice, compiling a version of the library that is about the same version as the one used in the target binary with different inlining options should yield a reasonably equivalent binary that could be used for analysation. However, in terms of the `glibc`, the vendor of the binary matters as the Linux distributions alter a string while building it which will reduce the matchings of this algorithm. To be precise, the offending string resides in the `_dl_close_worker` function: `TLS generation counter wrapped! Please report as described in <https://bugs.archlinux.org/>.\n`. Note that the URL changes depending on the origin of the compiled version of the library. For example, it would be the debian bug report URL for a version from the debian repos – the rest of the string would be identical. While at first sight this makes the algorithm slightly less effective, it is a huge benefit for the reverse engineer as such strings leak the origin of the used library. Usually, there are files with debug symbols available, thus increasing the total number of found subroutines, because a closer match of the development setup used to compile the library can be achieved.

It may not be necessary to get an object file suitable for static linking. As long as the constants and the string cross references are not altered, the matching will succeed. Still, one has to be careful, because it can be shown that `alignof()` evaluates differently compiling a relocatable object compared to a non-relocatable. While functionally equivalent, this leads to different strings when used in `assert()`, like in `glibc`s `malloc()`:

```
2391 assert ((unsigned long) (old_size) < (unsigned
         long) (nb + MINSIZE));
```

Figure 1: *Problematic `assert()` in `malloc.c`*

Compiling `glibc` 2.21, this assert evaluates differently depending on wether relocation is enabled or not:

●
```
nb + (unsigned long)(
  (
    (
      (__builtin_offsetof(struct
          malloc_chunk, fd_nextsize)) +
      (
        (2 * (sizeof(size_t)) <
            __alignof__ (long double) ?
          __alignof__ (long double) :
          2 * (sizeof(size_t))
        ) - 1)
      )
      & ~(
        (2 * (sizeof(size_t)) < __alignof__
            (long double) ?
          __alignof__ (long double) :
          2 * (sizeof(size_t))
        ) - 1
      )
    )
  )
)
```

compiling **without** relocation enabled

●
```
nb + (unsigned long)(
  (
    (
      (__builtin_offsetof (struct
          malloc_chunk, fd_nextsize)) +
      (
        (2 * (sizeof(size_t))) - 1
      )
    )
    & ~(
      (2 * (sizeof(size_t))) - 1
    )
  )
)
```

compiling **with** relocation enabled

Note however, that these listings are prettified versions of the actual string produced by `assert()` to enhance readability. The actual output has neither indentation nor line breaks.

Automatic binary analysis

This step will be applied to both the input object file and the target binary. Both I will refer to as "input file" hereinafter. Based on a pre-gernerated list of subroutines in the input file, every function's instructions will be examined. Such a list can be obtained using a debugger of your choice, for example IDA Pro or radare2.

In the following, $str(x)$ denotes the string retrieved from a given immediate that exists iff there is a string at the generated offset of x which can be determined using (**??**); $imm(x)$ is the immediate of the instruction; $op(x)$ the first byte of the opcode of the instruction. Furthermore, $callee(x)$ is the node N of the function for a given immediate x; N_c shall denote the currently examined function. Let us define the following three *inclusion rules*, which will determine if and to which list a given instruction immediate will be added. Function cross-references will also be resolved, however only near calls are resolved and relocated functions (from extern libraries, for example) are ignored. Note that these formal definitions only hold true for the Intel x86_64 Architecture. While the algorithm is not bound to any specific architecture, these definitions need to be adapted when applying to different architectures.

$$
\begin{aligned}
&\forall i \in I \in N_c : \\
&B \cup \{N_c, \text{callee}(\text{imm}(i))\} \iff && (2)\\
&\qquad i \in \{\texttt{call}\} \wedge \text{op}(i) = \texttt{0xE8} \\
&S \cup \{\text{str}(\text{imm}(i))\} \iff && (3)\\
&\qquad i \in \{\texttt{lea}, \texttt{mov}\} \wedge \exists \text{str}(x) \\
&C \cup \{\text{imm}(i)\} \iff && (4)\\
&\qquad i \in \{\texttt{and}, \texttt{or}, \texttt{xor}, \texttt{mov}\} \wedge \exists \text{imm}(i)
\end{aligned}
$$

Matching

There are many (Sub)Graph isomorphism[3] algorithms. Today, Ullmanns [?] algorithm is still one of the most popular ones used for exact graph matching, although it was originally developed for graph and subgraph isomorphism. [?][P. 121ff] compared it to other algorithms and coming to the conclusion that it is the most appropriate if the input graphs are small to mid-size (up to 500 nodes) and are unattributed.

Discussing graph isomorphism, it is also necessary to mention the Nauty algorithm [?]. Before checking the isomorphism, it first transforms the graph into a canonical form. While it is one of the fastest algorithms available, it has been shown that there are classes of graphs where its time complexity employs exponential time. Furthermore, it only provides exact graph isomorphism, being not able to solve the graph-subgraph isomorphism problem, as it partitions whole colored graphs for matching.

A third popular algorithm is the VF2 Algorithm [?], which is based on five *feasibility rules* and a state space representation (SSR) of the matching process. Essentially, the SSR used is a vector of nodes in the matching. This representation allows to check syntactic and semantic equivalence in one step, allowing a faster pruning of the search tree. Comparing VF2 to other algorithms shows that it has significantly better time and memory complexity than Ullmann and is for small randomly connected graphs faster than Nauty; on 2D mesh graphs, it outperforms Nauty if $|N| \geq 80$.

For this algorithm, I decided to rely on VF2, mainly for three reasons:

[3]An isomorphism is a function that maps the shape of two mathematical structures. From the ancient greek *isos* = equal and *morphe* = shape.

- It is fast - even if it is not the fastest algorithm, it always performs at least second best compared to Nauty and Ullmann [? , p. 1370].
- Callgraphs cannot be considered purely randomly connected, as there are functions which imply calls other functions afterwards, like `malloc()` and `free()` or `accept()` and `close()`. While these do not directly apply to `glibc`, as only a subset of such functions is actively used there, the general concept should be evident. Also, there are a lot of trivial, regular 2D mesh call graphs opposed to the number of huge non-regular ones like the callgraph of `vfprintf`. VF2 outperforms Nauty and Ullmann on graphs showing at least some kind of regularity [? , p. 1370].
- Its ability to check semantic and syntactic equivalence in one step, allowing the algorithm to check it semantic dependencies such as strings and cross references while checking syntactic equivalence.

Let $G_1 = (N_1, B_1)$ and $G_2 = (N_2, B_2)$ be two graphs that should be matched. A mapping $M \subset N_1 \times N_2$ is a mapping iff M is a bijective function that does not alter the branch structure of the two graphs. A state space representation s is introduced, with $M(s)$ describing the partial mapping of a state s. It contains only a subset of M. From $M(s)$, two subgraphs $G_1(s)$ and $G_2(s)$ can be derived, containing only those nodes already present in the partial mapping and the branches connecting them. These nodes, with their branches, univocally identify a subgraph of G_1 or G_2, respectively. Furthermore, $M_1(s)$, $M_2(s)$, $B_1(s)$ and $B_2(s)$ are defined as the nodes (and branches respectively) of $G_1(s)$ and $G_2(s)$.

Using this definition, a transition from a state s to a successor state s' is the simple addition of a pair (n, m) of matched nodes. Still, only a small subset of these states are *consistent*, in the sense that nothing impedes reaching a complete solution where no nodes are left unmatched. To further reduce the search space, a set of *k-lookahead-rules* is introduced, determining whether a consistent state can be reached after k iterations. These rules will hereinafter be called *feasibility rules*, to be consistent with the original paper.

The most general form of the feasibility function, which is true if the addition of a pair (n, m) satisfies all feasibility rules, taking also into account not only syntactical but also semantical equivalence, is the following:

$$F(s,n,m) = F_{syn}(s,n,m) \wedge F_{sem}(s,n,m)$$

In the initial state s_0, the matching M is empty, i.e. $M(s_0) = \emptyset$. In each intermediate step, the algorithm computes a set $P(s)$ containing the node pairs that are candidates to be added to the intermediate state s. This is done by first considering all nodes directly connected to $G_1(s)$ and $G_2(s)$ respectively. Let $T_n^{in}(s)$ denote the nodes with branches ending into $G_n(s)$, and $T_n^{out}(s)$ as the nodes with branches starting from $G_n(s)$. To employ the depth-first strategy, $P(s)$ will consist of all node pairs (n,m) with $n \in T_1^{out}(s)$ and $m \in T_2^{out}(s)$. If one of these two sets is empty, T_n^{in} shall be used instead. If one of these sets is empty too, the algorithm backtracks.

For the next step, the five feasibility rules are defined as follows. These check the syntactic equivalence: the first two check the consistency of adding a node pair to the state s, the rest is used to prune the search tree utilising a 1- and 2-look-ahead, respectively. $Pred(G,n)$ shall denote the set of predecessors of a node n in a graph G; $Succ(G,n)$ the set of successors of a node n in a graph G. Furthermore, we define $\widetilde{N}_n(s) = N_n - M_n(s) - (T_n^{in} \cup T_n^{out})$.

$$\begin{aligned}
&R_{pred}(s,n,m) \iff \\
&(\forall n' \in M_1(s) \cap \mathrm{Pred}(G_1,n) \exists \\
&\qquad m' \in \mathrm{Pred}(G_2,m) | (n',m') \in M(s)) \wedge \\
&(\forall m' \in M_2(s) \cap \mathrm{Pred}(G_2,m) \exists \\
&\qquad n' \in \mathrm{Pred}(G_1,n) | (n',m') \in M(s))
\end{aligned}$$

$$\begin{aligned}
&R_{succ}(s,n,m) \iff \\
&(\forall n' \in M_1(s) \cap \mathrm{Succ}(G_1,n) \exists \\
&\qquad m' \in \mathrm{Succ}(G_2,m) | (n',m') \in M(s)) \wedge \\
&(\forall m' \in M_2(s) \cap \mathrm{Succ}(G_2,m) \exists \\
&\qquad n' \in \mathrm{Succ}(G_1,n) | (n',m') \in M(s))
\end{aligned}$$

$$\begin{aligned}
&R_{in}(s,n,m) \iff \\
&(\left|Succ(G1,n) \cap T_1^{in}(s)\right| = \\
&\qquad \left|Succ(G2,m) \cap T_2^{in}(s)\right|) \wedge \\
&(\left|Pred(G1,n) \cap T_1^{in}(s)\right| = \\
&\qquad \left|Pred(G2,m) \cap T_2^{in}(s)\right|)
\end{aligned}$$

$$\begin{aligned}
&R_{out}(s,n,m) \iff \\
&(\left|Succ(G1,n) \cap T_1^{out}(s)\right| = \\
&\qquad \left|Succ(G2,m) \cap T_2^{out}(s)\right|) \wedge \\
&(\left|Pred(G1,n) \cap T_1^{out}(s)\right| = \\
&\qquad \left|Pred(G2,m) \cap T_2^{out}(s)\right|)
\end{aligned}$$

$$\begin{aligned}
&R_{new}(s,n,m) \iff \\
&(\left|\widetilde{N}_1(s) \cap \mathrm{Pred}(G_1,n)\right| = \\
&\qquad \left|\widetilde{N}_2(s) \cap \mathrm{Pred}(G_2,m)\right|) \wedge \\
&(\left|\widetilde{N}_1(s) \cap \mathrm{Succ}(G_1,n)\right| = \\
&\qquad \left|\widetilde{N}_2(s) \cap \mathrm{Succ}(G_2,m)\right|)
\end{aligned}$$

Additionally to these *syntactic* feasibility rules, a sixth *semantic* feasibility rule is introduced. For this rule, a compatibility relation $\approx$ between two nodes and their node/branch attributes is defined. Although for some applications $\approx$ may coincide with the equality relation, for our attributed call graph this is not the case. Using our designated node definition, the compatibility relation can be defined this way:

$$\begin{gathered}
n \approx m \iff \\
(\forall c \in C_n \, \exists \, c' \in C_m | c = c') \wedge \\
(\forall c \in C_m \, \exists \, c' \in C_n | c = c') \wedge \\
(\forall s \in S_n \, \exists \, s' \in S_m | s = s') \wedge \\
(\forall s \in S_m \, \exists \, s' \in S_n | s = s')
\end{gathered}$$

This yields the final semantic feasibility rule:

$$\begin{aligned}
&F_{sem}(s,n,m) \iff n \approx m \\
&\qquad \wedge \forall (n',m') \in M(s), (n,n') \in B_1 \Rightarrow \\
&\qquad\qquad (n,n') \approx (m,m') \\
&\qquad \wedge \forall (n',m') \in M(s), (n',n) \in B_1 \Rightarrow \\
&\qquad\qquad (n',n) \approx (m',m)
\end{aligned}$$

The matching is done in a brute-force manner, matching every graph from the object file with every graph from the target file. If a unique matching is found, each node in the target graph of the matching gets a name assigned based on the one in the matched object file graph. If more than one possible matching is found, the top node is added to a list of ambiguous nodes with the rest of the nodes in the matching remaining unchanged. For the following formal definition, let us define $vf2(G_1, G_2)$ as the VF2 matching function outlined above; $cg(n)$ as the call graph of the node N (that is, a sub graph of the whole library call graph). Furthermore, we define a set E, which holds the exact matches and a set A holding the ambiguous matches.

$$\begin{aligned}
&\forall n_o \in N_{obj} : \\
&\forall n_t \in N_{target} : \\
&A = A \cup \{\{n_o, n_t\}\} \iff \text{vf2}(\text{cg}(n_o), \text{cg}(n_t))
\end{aligned} \tag{5}$$

$$\begin{aligned}
&\forall \{n_o, n_t\} \in A | \ \nexists \{n_{o2}, n_{t_2}\} \in A | n_o = n_{o2} \land n_t \neq n_{t_2} : \\
&A = A \backslash \{n_o, n_t\}, E = E \cup \{n_o, n_t\}
\end{aligned} \tag{6}$$

Implementation issues

For comparison with other established symbol recovery methods, an implementation of this algorithm was made. This implementation is - alongside its sourcecode - freely available on GitHub[4].

In practice, it turned out that libraries may contain functions that contain neither callees nor suitable constants or strings. Some of these functions are called by an identifiable function, which in turn yields their name; however, there exist cases where such a mapping is not possible. These functions are not recoverable, for example, look at `secure_getenv()`, which has neither strings nor constants nor cross references as defined in this paper.

Likewise, there are cases in which nodes in the graph from the object file are considered equal as by our defined compatibility relation. In this case, function names can still be found, but it will most likely occur that there are multiple possibilities, if the function is not already identified in the graph of a unique function.

For this reason, the actual matching is split up in the implementation: first, match everything that can be uniquely identified, then iterate over the rest, this time allowing multiple matchings.

To save time upon matching, the analysis of the object file can be precomputed. A reasonable, json-based [**?**] data format has been developed. For examples, see the source code of the implementation.

Evaluation

For testing `glibc` function symbol recovery, a simple test program was created:

```c
int main()
{
    puts("Dummy");
    return 0;
}
```

Figure 2: *Source code of used test program*

This was then compiled with gcc and linked statically: `gcc -static -o dummy dummy.c`. Additionally, a stripped version, `dummy-stripped` was generated using `strip(1)`. Thereafter, a definition file was generated from the non-stripped version, or, depending on the algorithm used, the `libc.a` used for static linking. Subsequently, both definitions were applied to the stripped binary. The results are outlined in the following table.

definitions	linked against	find-magic	FLIRT (IDA)
`glibc 2.21` (arch linux)	`glibc 2.21` (arch linux)	376	233
`glibc 2.21` (arch linux)	`glibc 2.13` (debian-wheezy)	105	72

Figure 3: *Comparison of # of correctly recovered Symbols*

Not included in the list below, although worth mentioning, is the algorithms ability to give hints which function at a given offset *might* be. This is incredibly helpful for functions like `strcpy(3)`, where multiple versions utilizing different instruction sets (SSE/2/3/4, AVX, ...) exist, that all do the same job. They are usually indistinguishable; but all perform the same operation, so knowing the possible offsets for these functions yields what they do, even if the algorithm cannot correctly decide which one is at which offset. While this does not automate away manual reversing, it can be used as a starting point; which is a clear advantage over FLIRT[5].

Known limitations

As mentioned earlier, the algorithm fails to recover a function symbol if there are multiple possibilities for a function and the function is not part of a unique call graph. This may happen with subroutines that fulfill all of the following conditions:

1. Is not called within the library
2. Does not call anything

[4] `https://github.com/masterofjellyfish/findmagic`

[5] **F**ast **L**ibrary **I**dentification and **R**ecognition **T**echnology

3. Does not use any constants as defined in this paper

An example of such a function is outlined in the following figure:

```
.CapstoneX86Detail
    push rbp
    mov rbp, rsp
    mov rax, rdi
    add rax, 0x30
    leave
    retn
```

Figure 4: *Problematic function*

For reference, the original source code of this function might have looked like this:

```
cs_x86* CapstoneX86Detail(cs_detail *detail) {
        return &detail->x86;
    }
```

Figure 5: *Problematic function source code*

Excluding the immediate 0x30 from the constants makes sense. Altough we cannot match this function correctly, including it would make the algorithm much more version and compilation dependant:

- The value of the immediate is bound to the structure of the `struct` it operates on.
- The value can change between minor releases, as bugfixes are introduced
- Without knowing the target architecture, the offset of the x86 member in the `struct` is not predictable.

Theoretically, this algorithm is not bound to any specific architecture or operating system. However, it was developed and tested on GNU/Linux assemblies only. As long as a suitable method for resolving cross-references and extracting immediate operands can be found, the algorithm applies to any given architecture.

While false positives can happen, it is very unlikely on large functions, especially if they contain strings. Still, false positives are only a real problem if the real version of this subroutine is not included, while a false one is. Otherwise, one will not get a definitive match but hints instead.

Conclusion

In this paper, I presented a suitable algorithm for symbol recovery in statically linked binaries. For `glibc`, it performs better than the algorithm shipped with IDA Pro, a popular commercial disassembler. The symbol recovery algorithm also does not need the exact match of library version, compiler and compiler flags; a reasonably close assumption - maybe compiled from source for analysis - is enough in most cases.

References

[1] *Options That Control Optimization*. `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`.

[2] T. Bray. The javascript object notation (json) data interchange format, March 2014. RFC7159.

[3] T. Dullien and R. Rolles. Graph-based comparison of executable objects.

[4] B. P. Miller E. Jacobson, N. Rosenblum. Labeling library functions in stripped binaries. 2011.

[5] C. Eagle. *The Ida Pro Book: The Unofficial Guide To The World's Most Popular Disassembler*. No Starch Press, Inc., 2011.

[6] M. Van Emmerik. Identifying library functions in executable file using patterns. In *Software Engineering Conference, 1998. Proceedings. 1998 Australian*, pages 90–97, Nov 1998.

[7] Halvar Flake. Structural comparison of executable objects. 2004.

[8] C. Sansone L.P. Cordella, P. Foggia and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(10):1367–1372, Oct 2004.

[9] B. D. McKay. Practical graph isomorphism. 30:45–87, 1981.

[10] B. T. Messmer. Efficient graph matching algorithms, 1995.

[11] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976.