

An Approximate Maximum Common Subgraph Algorithm for Large Digital Circuits

Jochem H. Rutgers, Pascal T. Wolkotte, Philip K.F. Hölzenspies, Jan Kuper, Gerard J.M. Smit
University of Twente, Department of EEMCS
P.O. Box 217, 7500 AE Enschede, The Netherlands
j.h.rutgers@utwente.nl

Abstract—This paper presents an approximate Maximum Common Subgraph (MCS) algorithm, specifically for directed, cyclic graphs representing digital circuits. Because of the application domain, the graphs have nice properties: they are very sparse; have many different labels; and most vertices have only one predecessor. The algorithm iterates over all vertices once and uses heuristics to find the MCS. It is linear in computational complexity with respect to the size of the graph. Experiments show that very large common subgraphs were found in graphs of up to 200,000 vertices within a few minutes, when a quarter or less of the graphs differ. The variation in run-time and quality of the result is low.

I. INTRODUCTION

Simulation of hardware designs is a very important step in the ASIC development flow. Although transaction level models can be used for the initial design space exploration, the final detailed implementation of the whole design has to be simulated as well. Implementations of small (portions of) designs can be simulated and debugged well using a software based simulator. However, bit and cycle accurate simulation of a large ASIC leads to prohibitive simulation times of multiple hours to days. In a hardware-in-the-loop simulation, FPGAs can emulate the implementation with a frequency that is only one or two orders of magnitude lower than the ASIC to be realized. However, the architectures of tomorrow will not fit in today’s largest FPGAs.

In [1], a new FPGA-based simulation technique is introduced that uses the regular and repetitive structure of many-core architectures. Instead of programming the whole architecture in the FPGA at once, only a single core is instantiated and the individual cores are evaluated *sequentially*, reusing the same hardware. As all cells—where the general term *cell* denotes any distinct design element, e.g. a single processor core—in a homogeneous many-core architecture are identical, we only have to instantiate a single cell’s combinatorial functionality in the FPGA. For the sequential simulation, the state of each core, stored in the on-chip memory, can iteratively be loaded into and read from the instantiated hardware. Therefore, simulation of one clock cycle of an n -core system takes in principle n FPGA clock cycles. This simulation method reduces the FPGA resource requirements enormously and allows simulation of larger designs, without reconfiguring the FPGA.

To apply the same simulation method to heterogeneous

many-core architectures, the problem arises to find the similarities between the cells that make up the design. When the common logic is found, a ‘supercell’ can be constructed, which embodies all cells, without duplicating logic. Then, this supercell is instantiated in the FPGA, instead of all separate cells. This paper focusses on the algorithm to find the common logic of two cells, which can be used as basis for the construction of the supercell.

We assume that the algorithm is applied to two cells that have much logic in common. First, the netlists of both cells, e.g. in EDIF format, are converted into graphs. Then, the Maximum Common Subgraph (MCS) is determined—a well-known problem in graph theory. Since the cells can be very large, built of thousands of technology primitives, such as AND gates or flip-flops, exact algorithms cannot be applied. This paper presents an approximate MCS algorithm that is optimized for graphs representing netlists.

Our application uses the algorithm for simulation of large ASIC designs. However, the MCS algorithm can be used in a wider range of applications, for example FPGA run time reconfiguration optimization [2].

We continue this paper with an overview of existing algorithms to find the MCS. Those algorithms work fine for graphs with a small number of vertices. Given netlist graphs, which represent a netlist (as defined in Section III), an approximate MCS algorithm is presented in Section IV that can handle large graphs. The algorithm’s complexity and performance are discussed in Sections V and VI, respectively. In Section VII, we draw some conclusions.

II. RELATED WORK

Finding the MCS in random graphs is an NP-complete problem [3, 4, 5]. There are mainly two algorithms that can find the MCS in two arbitrary selected graphs: 1) The algorithm by *McGregor* that uses a state space representation, and 2) the *Durand-Pasari* algorithm that is based on an association graph and clique detection in it [3, 4, 6].

In [3], these algorithms are evaluated for various graphs. The performance depends on the *edge density* (η) of the given graphs, where η denotes the ratio between the number of existing versus all possible edges in the graph—a directed graph with n vertices can have n^2 edges at most. A graph with $\eta \approx 1$ is called *dense*, and one having $\eta \ll 1$ is

called *sparse*. For graphs having an $\eta < 0.05$, the McGregor algorithm is the fastest [3].

In literature, several papers report time measurements for finding the MCS. Graphs with about 100 vertices require a computation time in the order of hours [6]. With such run times for such small graphs, these algorithms are not a feasible solution to find the MCS of two large netlist graphs.

Three exact algorithms for dense, randomly connected graphs of at most 40 vertices are benchmarked in [6]. This requires calculation times ranging from minutes to hours. Many approximate algorithms exist to find the MCS faster than the two exact solutions [7]. However, their performance strongly depends on the type and properties of graph used. The size of the graphs analyzed is usually limited from 25 to 1000 vertices [3, 4, 6, 7].

Finding equivalences between digital circuits for verification use exact algorithms [8, 9], which are slow for large designs. In [2], the MCS is approximated by using a genetic algorithm, but matching graphs of 200 vertices still takes more than 9 hours. To the best of our knowledge, no other approximate MCS algorithm exists that is tailored to netlist graphs. Additionally, algorithms of other domains cannot be applied to netlist graphs, because of the specific graph properties that are used for optimization or heuristics.

III. NETLIST REPRESENTATION

Both ASIC and FPGA have a library with standard cells or *primitives*. An AND, OR, LUT, and D-FF are examples of such primitives. Graphs representing instances of such primitives, are defined as:

Definition Let $p \in P$ be an instance of a (technology) primitive with input ports I_p and output ports O_p . A *primitive graph* is defined as $g_p = (V_p, E_p, L_p)$, where g_p is a directed graph that represents p and

- $V_p = I_p \cup O_p \cup \{\mathbf{p}\}$ is the set of vertices (also called *nodes*) in the graph, where \mathbf{p} is a vertex that represents p ;
- $E_p = \{i \in I_p \mid \langle i, \mathbf{p} \rangle\} \cup \{\mathbf{p}, o \in O_p\}$ is the set of edges in the graph;
- $L_p(v \in V_p)$ is a function that assigns a label to every vertex, such that $L_p(\mathbf{p})$ is a label based on the type of the primitive p and $L_p(v \in I_p \cup O_p)$ is based on both the type of p and the name of the corresponding port.

The set P contains all instances of all primitives. In this paper, we only consider flattened netlists, so a *cell* consists of a subset of P . The *netlist* is the description (instances of primitives and wires) of this cell. This netlist can be described as a graph:

Definition Let c be a cell, built of primitive instances $P_c \subset P$, with input ports I_c and output ports O_c . A *netlist graph* is defined as $g_c = (V_c, E_c, L_c)$, where g_c is a directed graph that represents c and

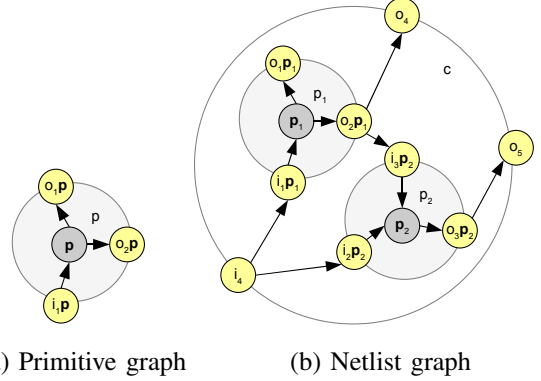


Figure 1: Graphs representing hardware

- $V_c = I_c \cup O_c \cup \bigcup_{p \in P_c} V_p$ is the set of vertices in the graph;
- $E_c = W \cup \bigcup_{p \in P_c} E_p$ is the set of edges, where W is the set of edges representing the wires in the netlist of c ;
- $L_c(v \in V_p)$ is a function that assigns a label to every vertex, such that $L_c(v) = L_p(v)$ when $v \in V_p$ for a given $p \in P_c$, otherwise $L_c(v)$ gives a label based on the port $v \in I_c \cup O_c$ it represents.

Fig. 1 visualizes a graph representing a primitive and depicts a netlist graph with two primitives.

Large netlists are converted into even larger graphs. For example, a Network-on-Chip (NoC) router, such as used in [1], is built of about 15k Xilinx Virtex-II FPGA primitives. Its equivalent graph has 80k vertices and 114k edges. The MCS algorithms discussed in Section II cannot handle such graphs within reasonable time. For this reason, we have developed an approximation algorithm, that is tailored to this specific type of graphs. Because they represent hardware, netlist graphs have nice properties:

- The graphs are very sparse. The edge density η ranges from $1.07 \cdot 10^{-5}$ for a large DSP processor, to $1.79 \cdot 10^{-5}$ for a NoC router and $4.57 \cdot 10^{-3}$ for a small FIFO.
- It is assumed that the MCS is a large part of the graphs.
- There are many different labels, based on multiple types of primitives that are used, and port names. Therefore, matching labels helps guiding the search.
- The top-level ports of the cells are highly similar, so a good and easy starting point of the MCS is known.
- Port vertices always have zero or one predecessor, which corresponds to one driver in the netlist at most. By exploiting this structural property, the search can be guided.
- Cycles in the graph are allowed, but no self-loops.
- Every vertex is (indirectly) connected to a vertex in O_c .

Each of these properties can be exploited in an algorithm to find a good approximation of the MCS fast, as outlined in the next section.

IV. ALGORITHM

In general, to find the MCS of graphs g_A and g_B —which represent cells A and B , respectively—a mapping M_{AB} is to be found from $v_A \in V_A$ of graph g_A onto the equivalent $v_B \in V_B$ in g_B . The resulting mapping M_{AB} defines the MCS, such that $M_{AB}(v_A) = v_B$ and therefore v_A and v_B represent the same vertex in the MCS. In order to construct M_{AB} , McGregor [10] determines for every $v_A \in V_A$ the *candidates* for mapping in V_B . Next, a backtracking search is used to find the optimal mapping from all vertices in V_A to their candidates in V_B .

Our (greedy) algorithm works as follows: 1) A best-first search is used to traverse the graph g_A . In every iteration of the search, one vertex of V_A is heuristically *chosen* in an attempt to find a mapping to a vertex in V_B . By having more candidates, the chance of choosing the wrong one increases. Therefore, vertices in V_A with a low number of mapping candidates in V_B are easy and should be processed first. 2) In contrast to McGregor, no backtracking is used. Therefore, an approximation of M_{AB} , denoted \hat{M}_{AB} , is obtained. 3) Of all candidates in V_B (if any), the best candidate is heuristically determined based on the *neighborhood* of v_A and its candidates: the candidate with the neighborhood that looks the most like v_A 's one, is chosen. 4) At the end of the iteration, the vertex v_A is *finished* and will not be chosen again, regardless whether a mapping has been found or not.

In every iteration of the best-first search, one vertex is taken out of the set D_A that contains *discovered* vertices of g_A . Then, the undiscovered neighbors of this vertex are discovered now and added to D_A . At the end of each iteration, the vertex that has been processed, is *finished* and added to the corresponding set F_A . Therefore, V_A can always be seen as the union of three disjoint sets: *finished* vertices F_A , *discovered* vertices D_A and *undiscovered* vertices U_A .

Algorithm 1 describes the complete algorithm. Subsequent sections will discuss the algorithm in more detail.

A. Initialization

One issue not addressed so far is the initialization (lines 1–3). The application of this algorithm provides natural starting points: the input and output vertices of the netlist graph, which are the top-level ports of the cell. Since g_A and g_B represent cells that are highly similar, they will certainly share a common interface to the outside world. In the worst case, the cell designs will still have a clock and/or reset input. Since the number of in- and outputs of a cell is assumed to be relatively small and not to scale (significantly) with the size of a design, the initial \hat{M}_{AB} is constructed by literal matching of port names of the I and O sets.

Given this starting condition, all vertices in I_A and O_A are finished and the set of all their neighbors are discovered. (The function *insert* is discussed in more detail below.) The rest of the vertices in V_A are undiscovered. However,

Algorithm 1: Approximate MCS algorithm

input : Graphs g_A and g_B representing cells A and B
output: The approximate MCS set defined by \hat{M}_{AB}

- 1 $F_A \leftarrow I_A \cup O_A$
- 2 $\hat{M}_{AB} \leftarrow \{(v_A : F_A, v_B : I_B \cup O_B) \mid L_A(v_A) = L_B(v_B)\}$
- 3 $D_A \leftarrow \text{insert}(\emptyset, (\bigcup_{f \in F_A} \text{neighbors}(f)) \setminus F_A)$
- 4 **while** $D_A \neq \emptyset$ **do**
- 5 $U_A \leftarrow V_A \setminus (F_A \cup D_A)$
- 6 $\langle D_A, d_A \rangle \leftarrow \text{choose}(D_A)$
- 7 $N_B \leftarrow \bigcup_{d'_A \in \text{neighbors}(d_A)} \hat{M}_{AB}(d'_A)$
- 8 $C_B \leftarrow \{c_B : V_B \mid L_B(c_B) = L_A(d_A) \wedge$
 $c_B \notin \text{range}(\hat{M}_{AB}) \wedge N_B \subseteq \text{neighbors}(c_B)\}$
- 9 **if** $C_B \neq \emptyset$ **then**
- 10 $c'_B \leftarrow \text{pick}\{b \in C_B \mid$
 $\neg \exists b' \in C_B \bullet \varepsilon(b', d_A) < \varepsilon(b, d_A)\}$
- 11 $\hat{M}_{AB} \leftarrow \hat{M}_{AB} \cup \{\langle d_A, c'_B \rangle\}$
- 12 $D_A \leftarrow \text{lower}(D_A, \text{neighbors}^2(d_A) \cap D_A)$
- 13 $D_A \leftarrow \text{insert}(D_A, \text{neighbors}(d_A) \cap U_A)$
- 14 $F_A \leftarrow F_A \cup \{d_A\}$

the initialization of U_A is done on line 5 to emphasize the invariance of this relationship between V_A , F_A and D_A .

B. Maintaining speculative order in D_A

The set of discovered vertices is implemented as an ordered list. The order of the list is ascending, based on the *expected* number of candidates a vertex will have once it is chosen. The function *insert* (lines 3 and 15) is used to add formerly undiscovered vertices to this list. Upon discovery, vertices are inserted at the front of the list.

The function *choose* (line 6) takes the first vertex in the list D_A and calculates the actual number of candidates it has. When the actual number is larger than the expected number, the element is reinserted in the list, ordered by its updated expectancy.

When a new mapping is found for a vertex (line 13), its mapped vertex c'_B in graph g_B cannot become a candidate ever again. The added mapping influences the set of candidates C_B of other previously discovered and not yet finished vertices. The expected candidate number of those vertices must thus be lowered. Finding precisely all those vertices for which the estimate must be lowered is an expensive operation. Instead, the neighbors of neighbors of the newly mapped vertex are all lowered, which is fast and sufficiently accurate. Since expectancies can thus be lowered too often for a vertex, this is where the expected number may become inaccurate again, which is corrected later by *choose*. This decrement of the expected number—and the appropriate reordering of the list—is the purpose of the function *lower* (line 14).

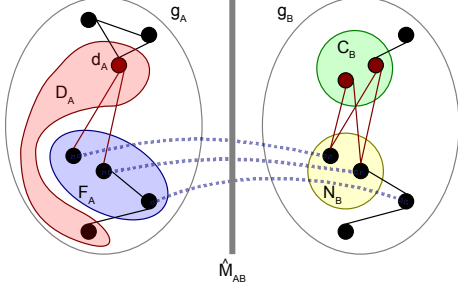


Figure 2: Sets of the algorithm's inner loop

C. Candidacy and choice

The set of candidates for a vertex is calculated by *choose*. For space considerations, this most significant part of *choose* is shown on lines 7–9, instead of treating the entire function separately. Adjacent vertices in the MCS are (obviously) adjacent in both g_A and g_B . Also, d_A is chosen from the neighborhood of the previously mapped vertices. The candidates for d_A must be neighboring vertices (in g_B) of all vertices that d_A 's neighbors are mapped to.

The set of all vertices that d_A 's neighbors are mapped to, is called N_B (line 7). The set of candidates for d_A is called C_B (lines 8–9). To be a candidate, a vertex must satisfy three constraints: it must have the same label as d_A (line 8); it may not already be mapped, where *range* gives all mapped vertices of V_B ; and it must be a neighbor to all vertices in N_B (both on line 9). Fig. 2 gives a graphic representation of the relations between d_A , N_B and C_B for any iteration of the algorithm.

When there are no candidates for a mapping of d_A , it is assumed not to be part of the MCS and not taken into further consideration (removed from D_A on line 6 and added to F_A on line 16). When there *are* candidates for a mapping (lines 10–15), one single candidate $c'_B \in C_B$ must be chosen. The function *pick* (line 11) chooses a vertex arbitrarily from a set of vertices. When there is exactly one candidate, the choice of c'_B is that one candidate. When there are multiple candidates, the added heuristic of edit distance is used to choose. This works as follows:

Definition The *extended neighborhood* $\nu_r(v)$ is the set of vertices that are reachable from vertex v in at most r steps:

$$\nu_r(v) = \bigcup_{i=1}^r \text{neighbors}^i(v) \quad (1)$$

Definition The *edit distance* ε of the extended neighborhoods of two vertices v_a and v_b is defined as

$$\varepsilon_r(v_a, v_b) = \sum_{l \in L_A \cup L_B} |\sigma(\nu_r(v_a), l) - \sigma(\nu_r(v_b), l)| \quad (2)$$

where $\sigma(V, l)$ is a function that returns the number of occurrences of label l in the set of vertices V .

Only the candidates in C_B with a minimal edit distance to d_A are chosen. If there are multiple candidates that all have the same (minimal) edit distance, *pick* chooses one arbitrarily.

V. COMPLEXITY

Finding the MCS is NP-complete. In [5], it is shown that the complexities for two graphs of size m and n are for example $O(1.19^{mn})$ for a general purpose algorithm, $O(m^{n+1}n)$ for an optimized backtracking algorithm and $O((m+1)^n)$ for the clique branching algorithm [5]. Approximate algorithms are of polynomial time complexity, which depends on the optimization techniques based on specific graph properties. In this section, the complexity of our approximate MCS algorithm is evaluated.

In graph theory, the edge density of (netlist) graph g_A is defined as $\eta_A = \frac{|E_A|}{|V_A|^2}$. However, the number of edges is bounded such that $|E_A| \propto |V_A|$. This can be explained by investigating the structure of the netlist graph.

Since port vertices can only have one predecessor, the number of edges in g_A is limited to the sum of: the number of output ports O_A ; the number of input ports of all primitives in P_A ; and the edges of g_p of all $p \in P_A$. $|I_A|$ and $|O_A|$ do not scale (significantly) with the size of the graph. The number of edges of the primitive graph is limited by the number of ports, which has a maximum value that is defined by the technology library. Therefore, $|E_A|$ only scales with the vertices of all $p \in P_A$ which leads by definition (see Section III) to $|E_A| \propto |V_A|$. Hence, $\eta_A \propto \frac{1}{|V_A|}$.

By definition, the average degree $\bar{\delta}_A$ of a vertex in g_A multiplied by the number of vertices equals twice the number of edges in g_A . Summarizing, η_A can be written as:

$$\eta_A = \frac{|E_A|}{|V_A|^2} = \frac{\frac{1}{2}\bar{\delta}_A|V_A|}{|V_A|^2} = \frac{\frac{1}{2}\bar{\delta}_A}{|V_A|} \quad (3)$$

Since $\eta_A \propto \frac{1}{|V_A|}$ and eq. (3), $\bar{\delta}_A$ has to be constant. So, $\bar{\delta}_A$ does not depend on the size of the graph.

Given these equations, the complexity of the complete algorithm can be determined. The algorithm has been implemented in Java, using the `HashMap` and `HashSet` as basic data structures. Both have constant time complexity for all basic operations. The list D_A is implemented as a map from the expected number of candidates to a set of vertices. Therefore, operations on D_A are constant in complexity. For every step in the algorithm, the complexity is determined:

- Every vertex in V_A is visited once: costs m .
- The function *neighbors* ^{r} iterates recursively with depth r over the edges, resulting in $\bar{\delta}^r$ neighbors: costs $\bar{\delta}^r$.
- N_B is the mapping for all neighbors and costs $\bar{\delta}_A$ to find.
- To find C_B , the algorithm iterates over all vertices in N_B . N_B is of size $\bar{\delta}_A$, so the costs for lines 8–9 are $\bar{\delta}_A \bar{\delta}_B$.
- To pick c'_B , we determine the ε for all vertices in C_B . The costs for calculating the ε of a single vertex pair is $\bar{\delta}_A^r + \bar{\delta}_B^r$ and the total costs for line 11–12 equals $\bar{\delta}_A \bar{\delta}_B (\bar{\delta}_A^r + \bar{\delta}_B^r)$.

- *lower* decrements in D_A $\bar{\delta}_A^2$ vertices, which costs $\bar{\delta}_A^2$.
- *insert* inserts on average $\bar{\delta}_A$ vertices in D_A at a fixed level, with a total cost of $\bar{\delta}_A$.
- *choose* takes the first element from D_A and determines its number of candidates. Every time the actual number of candidates is higher than expected, it is reinserted in D_A . In worst case, *lower* decremented all vertices too soon and will all be reinserted. In that case, in every iteration, $\bar{\delta}_A^2$ elements are reinserted, on average. The total costs for *choose* are $\bar{\delta}_A^2 + N_B + C_B$.

Since $\bar{\delta}$ and r are constant, the total complexity is:

$$\begin{aligned}
& \Theta(m(\text{choose} + N_B + C_B + c'_B + \text{lower} + \text{insert})) \\
&= \Theta(m((\bar{\delta}_A^2 + N_B + C_B) + N_B + C_B + c'_B + \bar{\delta}_A^2 + \bar{\delta}_A)) \\
&= \Theta(m(2\bar{\delta}_A^2 + 3\bar{\delta}_A + 2\bar{\delta}_A\bar{\delta}_B + \bar{\delta}_A\bar{\delta}_B(\bar{\delta}_A^r + \bar{\delta}_B^r))) \\
&= \Theta(m) \tag{4}
\end{aligned}$$

Concluded, the algorithm is linear in computational complexity, only with respect to the size of graph g_A .

The memory complexity is easier to calculate. The algorithm holds track of $|U_A| + |D_A| + |F_A| = m$ vertices. The number of mappings is at most m . The largest temporary variable contains the neighborhood, which is $\bar{\delta}^r$. So, the memory complexity is also $\Theta(m)$.

VI. EXPERIMENTS

This section will address two aspects of the presented algorithm: quality of the result and performance of the algorithm. In order to investigate the properties, experiments have been conducted on randomly generated graphs and graphs of real netlists.

The quality of the algorithm indicates whether the size of \hat{M}_{AB} is close to the size of the MCS. However, due to the extreme processing time for graphs of thousands of vertices, the MCS cannot be determined. The upper bound $\Psi(g_A, g_B)$ of $|\hat{M}_{AB}|$ can be determined easily by omitting the structure and counting the number of occurrences of each label in the graph as follows:

$$\Psi(g_A, g_B) = \sum_{l \in L_A \cup L_B} \min(\sigma(V_A, l), \sigma(V_B, l)) \geq |\hat{M}_{AB}| \tag{5}$$

The *quality* of the result is defined as $\frac{|\hat{M}_{AB}|}{\Psi(g_A, g_B)}$. If the quality is 1, it can be said that: the upper bound equals the size of the MCS *and* the MCS has been found—a perfect match. The *performance* is simply the time it takes to find the \hat{M}_{AB} , which is expected to be linear with respect to $|V_A|$, as shown in Section V.

To investigate the strong and weak points of the algorithm, random netlists with specific properties have been generated in Section VI-A. These graphs are constructed, such that the MCS is known and the quality is reliable. Section VI-B shows the result of the algorithm on real netlists, where the MCS is not known, but the quality is still high.

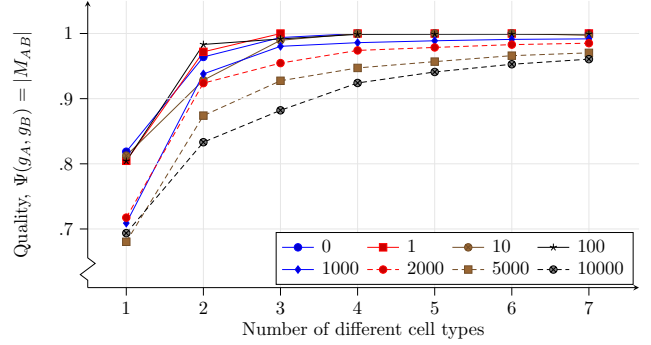


Figure 3: Quality of different types of random netlists with 20,000 cells after deletion of n cells

A. Random netlists

The randomly generated netlists are built of a specific number of different types of primitive cells, which are randomly connected, and have a specified size and number of input and output ports. The primitives themselves are also randomly generated, having 1 to 5 input and 1 or 2 output ports. Every test is repeated 100 times with different random graphs with the same parameters. The variation of the quality of the result is due to randomly generated cells and netlists during the setup of the test and the heuristics of the proposed algorithm during the test.

Several tests have been performed. In the first test, netlists of 20,000 cells—which results in a graph of about 100,000 vertices—are generated with 1 to 7 different cell types. The total number of cells is randomly distributed over all available different types. Next, n random cells (and cells that thereby get unconnected) are removed from the netlist, where n ranges from 0 to 10,000. The algorithm is applied to the original g_A and the resulting smaller graph g_B . By this setup, the extended neighborhood differs only because vertices are missing in g_B . Additionally, the upper bound Ψ exactly equals the size of the MCS.

Fig. 3 shows the results of this experiment. For example, the figure shows that when the netlists is built of only one type of cell and 10,000 cells are deleted, it results in an average quality of .69. This low quality is because *pick* cannot determine which is the best mapping based on the extended neighborhood; since there is only one type of cell, all neighborhoods look the same. When more different cell types are used in the netlist generation, the quality improves. When having 7 types or more, the average quality is .96 or higher, regardless of how many cells are removed from the netlist.

The next test is like the previous one, except that after n cells have been deleted, n cells were inserted, randomly chosen from same set of types. Then, all open ports are randomly reconnected. Therefore, $\frac{n}{|P_A|}$ of the netlist is replaced by roughly the same set of cells, but the structure

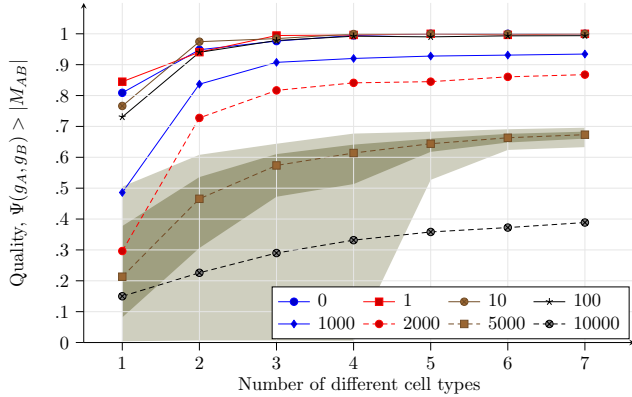


Figure 4: Quality of different types of random netlists with 20,000 cells after reinsertion of n cells

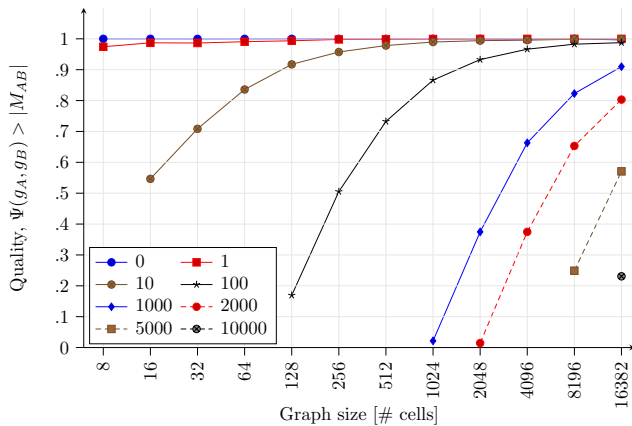


Figure 5: Quality of different sizes of random netlists with 5 cell types after reinsertion of n cells

has changed. In contrast to the previous test, the extended neighborhood differs also by structure and additional vertices, so it is harder to find the correct c'_B .

Fig. 4 shows the results. The quality is quite low, because the upper bound is too high: the structure and therefore also the MCS differ, but the set of labels did hardly change. Hence, for $n = 5000$, the upper bound is expected to be about $\frac{5000}{20000} = .25$ too high. Compensating for this effect, the resulting ‘normalized’ quality would be about $\frac{.67}{1-.25} = .90$ instead of $.67$. Similarly, for $n = 10000$ and 7 types, it is $.78$.

Fig. 4 also indicates the variation in results for $n=5000$. The light area indicates the highest and lowest measured quality and the dark area is the standard deviation of all values above and below the average quality added to the average. For other n , a similar converging trend is found. Concluded, having more types of cells results in a higher quality and lower variation because there are less candidates, so choosing c'_B gets easier.

The effect of having different sizes of netlists with 5 cell

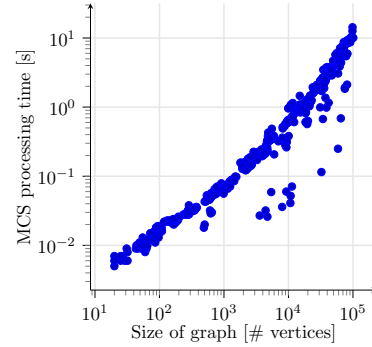


Figure 6: Required processing time for random graphs

types is shown in Fig. 5. For this experiment, netlists have been generated of 8 to 16,382 cells, where n cells have been replaced, like in the previous experiment. Therefore, the upper bound is expected to be too high, again. For example, the case that $n=2000$ cells are replaced of a netlist of 8,196 cells, the measured quality is $.65$ and $.86$ when normalized. So, when a quality of more than $.8$ is acceptable, about a quarter of the netlist can differ at most.

From other experiments it turns out that the number of input and output ports or extreme connectivity, such as a clock or reset signal, do not seem to influence the results as presented in this section.

In Fig. 6, the processing time is plotted of the same experiments as used in Fig. 5. The figure shows a slightly curved line. Since the algorithm is shown to be linear, it is likely that run-time effects influence the measurements, such as costly just-in-time compilation by the JVM for small graphs or increasing number of hash collisions for large graphs. The points below average correspond to the points with a low quality in Fig. 5; a very small MCS is found, which result in a low run time.

Concluded, the algorithm works well on netlist graphs having a large MCS and many types of cells, which corresponds to the properties as listed in Section III.

B. Real netlists

The tests were conducted on a dual core Intel Pentium 4 3.20 GHz. The algorithm itself uses one thread, but the Java VM has several other threads, for example for the garbage collector. For every test, the Java VM stays below 1 GB of memory usage. The range r for the neighborhood is set to 5. Based on these tests, $\bar{\delta} \approx 2.8$ for every graph.

Fifteen pairs of netlists are taken from existing designs. The (manual) choice of pairs is such that the resulting test set covers a wide range of designs, strongly differing in structure, functionality and size. These designs were not specifically built or modified for this experiment, instead they were taken from VHDL designs we had available. The VHDL code has been synthesized in a normal fashion,

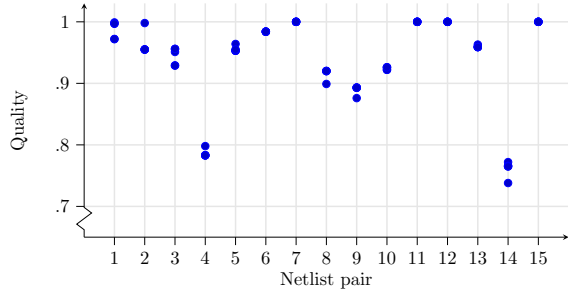


Figure 7: Quality of the result of real netlists

# component	$ V_A $
1. FIFO ₁	6,693
2. ALU ₁	2,526
3. crossbar	233
4. Round-Robin	145
5. FIFO ₂	5,640
6. interconnect	12,111
7. ALU ₂	16,701
8. proc. part ₁	36,106
9. proc. part ₂	36,364
10. AGU	2,081
11. RAM	1,328
12. DSP proc.	191,766
13. ALU ₃	3,601
14. busses	8,591
15. register file	3,040

Table I: 15 real netlists for test

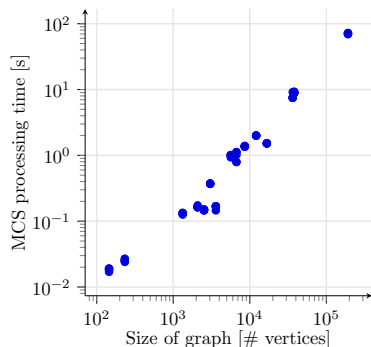


Figure 8: Performance of the algorithm on real netlists

exported to EDIF format and converted to graphs. The pairs were designed to be the same, e.g. by choosing two ALUs of a DSP processor that has five, but are different mainly due to synthesis (boundary) optimizations. Table I briefly summarizes the fifteen designs and the number of vertices of the corresponding graph g_A .

Fig. 7 shows the quality of the result of these pairs. Every test has been repeated five times. The figure shows that the *variation* in quality between different runs of the same test is low. This indicates that the algorithm is ‘stable’ and does not depend (much) on the random order of the sets. Based on previous experiments, it must be concluded that test 4 and 14 have a relatively low quality due to an upper bound that is too high; the labels in the graph are largely the same, but the structure is not.

Fig. 8 shows the required time to find the \hat{M}_{AB} for the same set of tests. Although the tests consists of various netlist graphs, the figure shows that the algorithm is close to linear in computational complexity. The data for this figure is collected in the same run as for Fig. 7. The variation in run time for different runs is also small.

VII. CONCLUSION

This paper presented an approximate MCS algorithm, specifically for directed, cyclic graphs representing netlists. Because the graphs represent hardware, they have a number of nice properties: they are very sparse; have many different

labels; and most of the vertices have only one predecessor. The algorithm uses two heuristics: every vertex in graph g_A is visited only once, in order of the number of mapping candidates; and its mapping is chosen based on the equivalence of the neighborhood. The computational complexity scales linearly with the size of the graph.

The algorithm has been implemented in Java and tested with random netlist graphs of up to 100,000 vertices and real netlist graphs of 145 to 191,766 vertices. The algorithm performs well when the MCS spans about three quarters of the graph or more, consisting of five different types of cells or more. The variance in quality of the result and the run time is low. For all tests with real netlists, large common subgraphs were found, which approximate or equal the MCS within 72 seconds for the largest graph.

REFERENCES

- [1] P. T. Wolkotte, P. K. F. Hölzenspies, and G. J. M. Smit, “Fast, accurate and detailed NoC simulations,” in *Proc. First ACM/IEEE International Symposium on Networks-on-Chip*, P. Kellenberger, Ed. Los Alamitos, CA, USA: IEEE Computer Society Press, May 2007, pp. 323–332.
- [2] H. Fröhlich, A. Kosir, and B. Zajc, “Parallel genetic algorithm for optimizing run-time reconfigurable circuits,” in *Proc. 10th Mediterranean Electrotechnical Conference MELECON 2000*, vol. 1, 29–31 May 2000, pp. 37–40.
- [3] H. Bunke, P. Foggia, C. Guidobaldi, C. Sansone, and M. Vento, “A comparison of algorithms for maximum common subgraph on randomly connected graphs,” in *SSPR&SPR 2002, LNCS 2396*, 2002, pp. 123–132.
- [4] D. Conte, P. Foggia, and M. Vento, “Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs,” in *Journal of Graph Algorithms and Applications*, 2007, vol. 11, no. 1, pp. 99–143.
- [5] W. Henry Suters, F. Abu-Khzam, Y. Zhang, C. Symons, N. Samatova, and M. Langston, “A new approach and faster exact methods for the maximum common subgraph problem,” in *Computing and Combinatorics*, ser. LNCS. Springer Berlin / Heidelberg, 2005, vol. 3595/2005, pp. 717–727, ISBN: 978-3-540-28061-3.
- [6] Y. Wang and C. Maple, “A novel efficient algorithm for determining maximum common subgraphs,” in *Proc. Ninth International Conference on Information Visualisation*, 6–8 July 2005, pp. 657–663.
- [7] J. Raymond and P. Willett, “Maximum common subgraph isomorphism algorithms for the matching of chemical structures,” *J Comput Aided Mol Des*, vol. 16, no. 7, pp. 521–533, Jul 2002.
- [8] W. Kim and H. Shin, “Hierarchy Restructuring for Hierarchical LVS Comparison,” in *VLSI Design*, 1999, vol. 10, no. 1, pp. 117–125.
- [9] M. Ohlrich, C. Ebeling, E. Ginting, and L. Sather, “Subgemini: Identifying subcircuits using a fast subgraph isomorphism algorithm,” in *Proc. 30th Conference on Design Automation*, 1993, pp. 31–37.
- [10] J. McGregor, “Backtrack search algorithms and the maximal common subgraph problem,” in *Software—practice and experience*. John Wiley & Sons, Ltd., 1982, vol. 12.