

University of Twente

EEMCS

Formal Methods and Tools

FPGA-on-FPGA emulation using node disjoint subgraph homeomorphism

MSc Thesis

Pim van Leeuwen (s1602772)

Technical Supervisor

dr. W. Kuijper

Academic Supervisors

dr. ir. R. Langerak

H.H. Folmer MSc.

UNIVERSITY
OF TWENTE.



Abstract

FPGAs allow reconfiguration of its logic at any point after production. The result is that they are effective at prototyping application-specific integrated circuits, updating the internal logic while in the field and at low-cost low-quantity use cases. To optimise these processes, it is crucial to properly educate engineers in the implementation of FPGA programs and the FPGA compilation process. Traditional FPGA programming pipelines involve a computationally expensive (NP-hard) place & route process that slows down iterations of FPGA programs and hinders the educational process. We propose a virtual environment in which place & route is performed manually in which the student learns about the intricacies of place & route and in which compilation is linear. To this end, we require emulation of a virtual FPGA on a physical, concrete FPGA. In the proposed research, we find such an emulation using an algorithm that solves a variant of subgraph isomorphism. This algorithm aims to find emulations in as many cases as possible and to exploit the hierarchy of FPGAs to speed up computation. The expected result is a software package that computes emulators which each output a program for a concrete FPGA provided with a program for a virtual FPGA.

CONTENTS

1	Introduction	3
2	Background	5
2.1	FPGAs	5
2.1.1	Lookup tables	6
2.1.2	Registers	7
2.1.3	Logic cells	7
2.1.4	Routing	8
2.1.5	Pins	8
2.2	FPGA compilation	9
2.2.1	Literature	10
2.3	FPGA simulation	11
2.4	FPGA emulation	11
2.5	Graph theory	12
2.5.1	Literature	13
3	Objectives	18
4	Models	19
5	Algorithm	21
5.1	Baseline: ndSHD2	21
5.2	Parameters	23
5.3	Optimisations	24
5.3.1	Refusing long paths	25
5.3.2	Runtime Pruning	25
6	Pruning	26
6.1	Domain filtering	27
6.1.1	Labels and neighbours	27
6.1.2	Free neighbours	27
6.1.3	Reachability of matched vertices	28

6.1.4	Reachability of neighbourhood	28
6.2	Pruning methods	29
6.2.1	ZeroDomain pruning	29
6.2.2	AllDifferent pruning	30
6.3	When to apply	30
6.3.1	Runtime calculation	30
6.3.2	Caching domains - incremental domain calculation	30
6.3.3	Parallel calculation	31
7	Use case: Lattic ECP5	32
8	Performance experiments	36
9	Software design	39
9.1	Architecture	39
9.2	Manual	39
10	Conclusion	40
11	Discussion	41
12	Future Research	42
	Appendices	43
.1	History of subgraph isomorphism algorithms	44

1 INTRODUCTION

Field Programmable Gate Arrays (FPGAs) play a significant role in the semiconductor industry. Because companies can configure the logic of these devices after mass hardware production, they can effectively use them for prototyping and emulation of Application Specific Integrated Circuits (ASICs). These are chips that are custom made for specific purposes and cannot be reconfigured. FPGAs can also be used to reconfigure program logic effectively, even while the hardware is in use. Moreover, the high computing performance for parallel calculations yielded by a small chip makes an FPGA suitable for sensor systems[22], cryptography[54, 42, 29], digital signal processing[21], protocol implementations[46] and other types of high-performance computing[31].

We should adequately educate the engineers that implement FPGA programs to exploit their potential in this variety of problem areas. Usually, this is done by having students implement programs in a Hardware Description Language (HDL) such as VHDL or Verilog, compile it using FPGA vendor software to bit-level code that is written to FPGA hardware and execute it. This type of compilation takes a significant amount of time as the place & route process required for compilation are both NP-complete problems[19]. Another option is to simulate FPGA programs instead. Simulation circumvents the place & route process. However, simulation software run on a CPU has to simulate each component sequentially for each CPU core, resulting in enormous performance loss.

Another technique is modelling the semantics of FPGA components in an HDL module. The engineer can then simulate the execution of an FPGA program by providing both the input *and* configuration of these components as (stored) inputs to the hardware. An example of this is illustrated in Section 2.3 and Figure 2.6. The trade-off here is that the hardware uses many resources for the simulation of each component. Because of this increased resource consumption, the performance of FPGA programs decreases as well.

Techniques that require students to implement FPGA programs in HDLs furthermore abstract away the process of place & route, a process that significantly affects the performance of the hardware implementation of the program. Students should be aware of this process and its underlying operations to build a bottom-up understanding of the entire FPGA engineering pipeline.

We propose an education environment where students implement program logic and place

& route manually in a simple, virtual FPGA. Performing the NP-complete place & route manually allows the rest of the compilation process to take linear time. This compilation process specifically compiles an FPGA-configuration meant for a virtual FPGA layout to a configuration compatible with a concrete one.

This way, students can circumvent the time-consuming compilation from HDLs to FPGAs, resulting in faster iterations and improved learning experiences. Furthermore, students can use this environment to learn the inherent difficulty of place & route before they use tools that encapsulate this process in a ‘black box’. They will have to do this when they implement more complex structures in the virtual environment. We can achieve this emulation by calculating a mapping from the configuration of a virtual FPGA to the configuration of a real-life FPGA board (concrete FPGA). While the format of an FPGA configuration is usually kept secret by hardware vendors[26, 59, 4], some FPGA boards have been reverse engineered to reveal the underlying format[55]. Once this technique has been implemented and tested on those FPGAs, It can be put in practice in lab sessions of colleges and universities. Moreover, vendors can implement it for undisclosed FPGA designs and configuration formats as well for educational purposes as well.

This research applies an algorithm for the graph problem ‘node disjoint subgraph homeomorphism’: an embedding of some source graph in a target graph, where addition of intermediate vertices is permitted. Applying this algorithm to graph models of FPGA provides such a mapping. We furthermore improve this algorithm using various methods from the domain of subgraph isomorphism (a similar graph problem), evaluate the different settings/improvements of this algorithm and how practical this algorithm is for educational use cases.

Another use case of such mappings is synthesis of the same configuration to many hardware FPGA devices. Using our technique, synthesis of some FPGA program only has to take place once (to a virtual FPGA) before suitable configurations can be retrieved for many hardware architectures. Other applications may also benefit from this research when viewing partial FPGA emulation as a form of pre-compilation: performing FPGA compilation as much as possible without the knowledge of some aspects of the configuration. If this research is extended to include other forms of partial compilation, it may be used to improve the speed of general iterations of FPGA development as well.

2 BACKGROUND

2.1 FPGAs

The computing hardware most people are familiar with is CPUs. Manufacturers incorporate them in every desktop pc, laptop, and most mobile devices. CPUs are very flexible and efficient- which is why they are the de facto standard for any computation task. In CPU computation, an integrated circuit (a processor) iteratively reads an instruction from a RAM module (in the form of encoded bits), performs the instruction, and then continues to read the next instruction. The instructions are not embedded in the circuit of the CPU itself.

FPGAs are different from CPUs, as they do not store the programs they execute in RAM- they instead configure them in the (highly parallel) logic of the circuit itself. Configuring an FPGA to execute a specific program entails loading a configuration file onto the hardware and restarting the FPGA such that it reconfigures its logic. The hardware will then perform the configured logic on the input it receives via IO pins.

Because each logic cell performs logic independently, FPGAs can perform computations highly parallel and without delays from loading instructions. This computation process implies that FPGAs are very efficient at executing individual programs, specifically concurrent

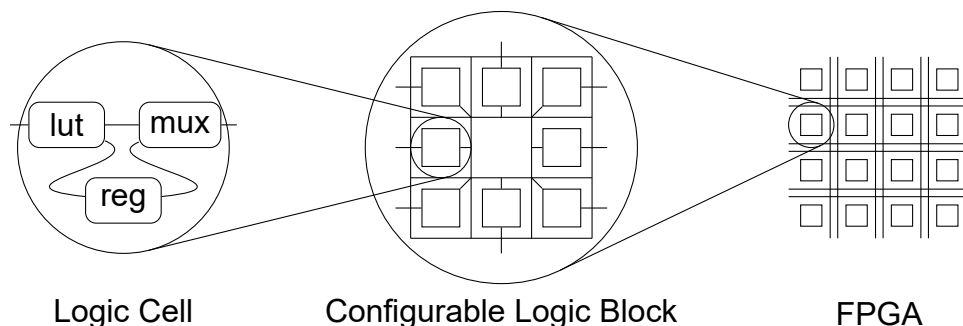


Figure 2.1: The hierarchy of a typical FPGA. A typical FPGA mostly consists of Configurable Logic Blocks (CLBs) and a CLB mostly consists of logic cells. The way logic cells and CLBs are connected may be different for each FPGA architecture.

p	q	$p \text{ XOR } q$
F	F	F
F	T	T
T	F	T
T	T	F

Figure 2.2: A truth table that shows the result of an XOR operation

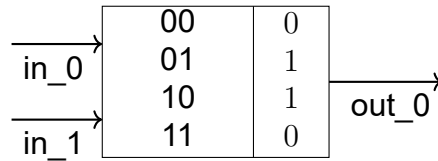


Figure 2.3: A Lookup Table (LUT) in which an XOR-operation is configured

ones. Reconfiguring an FPGA is, however, a relatively expensive operation. Therefore, FPGAs are unsuitable for changing from program to program, as a CPU does when running an operating system.

FPGAs perform execution using components such as lookup tables, registers, and special-purpose modules¹. These components are physical structures on the circuit. In the next sections, we will discuss these modules.

2.1.1 Lookup tables

Any boolean logic formula can be expressed in the form of a truth table, such as in Figure 2.2. The leftmost column of a truth table specifies all possible combinations of T and F for all inputs; the rightmost column then specifies what output that specific logic function would give. Each distinct combination of T and F in the rightmost column corresponds to a different boolean formula. FPGAs execute logic in the form of Lookup Tables (LUTs), which model the evaluation of a truth table, but with ones and zeroes instead of T and F : for every combination of ones and zeroes in the input, the LUT stores what output it should give. The compilation software provided by the FPGA vendor can reconfigure these outputs. This way, any boolean formula with the appropriate number of variables can be implemented with a lookup table. Figure 2.3 shows a lookup table that has two input bits and one output bit. This lookup table is configured to perform an XOR-operation. Lookup tables can have input and output consisting of any number of bits, depending on the FPGA design.

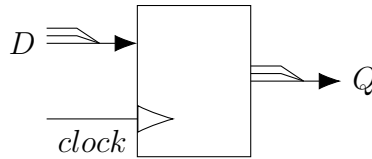


Figure 2.4: Traditional representation of a register. Note that input D and output Q may consist of multiple wires.

2.1.2 Registers

Programs require intermediate data storage to perform any computation that is more complex than finite logic or fixed-point calculations.

FPGAs use registers (or D flip-flops) for this purpose (see Figure 2.4). Registers can store a small collection of bits, depending on the FPGA design. When a register's input *clock* changes from 0 to 1, the contents of the register are replaced by the value of the input D , and the output Q takes this value. The output stays constant until the clock changes from 0 to 1 again when D has a different value.

FPGAs usually have a global clock- a wire which signal constantly changes between 0 and 1 that is connected to all registers in the hardware. The frequency of this clock is chosen such that all signals are guaranteed to be stable when the clock becomes 1. This stability is very convenient for programmers, who do not have to calculate the time it takes for a signal to propagate through a wire. The implication is that each circuit combining only lookup tables that end in the D-input of a register takes the same amount of time.

If the FPGA program has longer chains of lookup tables, then it takes longer for the output signal to stabilize. The vendor software accommodates for this by setting the global clock at a lower frequency. Since programs on FPGAs are highly parallel, there are likely some parts of the calculation that do not require a lower clock speed and can cause slowdown because of this. In these scenarios, adding registers to some parts of an FPGA program can improve performance if it allows the global clock speed to be higher.

2.1.3 Logic cells

A typical logic cell is a combination of one or more lookup tables, a register, and a MUX (a 3-input gate that outputs a copy of the first or second input, depending on the value of the third input). The contents of the lookup tables can be configured to perform any logic operation, and the value of the third MUX-input can be configured to specify whether the computation is clocked/synchronous or combinatorial/asynchronous. A logic cell is shown in Figure 2.5. Unfortunately for us, vendors have different logic cell designs, meaning we cannot generalize this example.

¹These modules are used for efficient storage or calculation of specific functions. We will disregard them in this research.

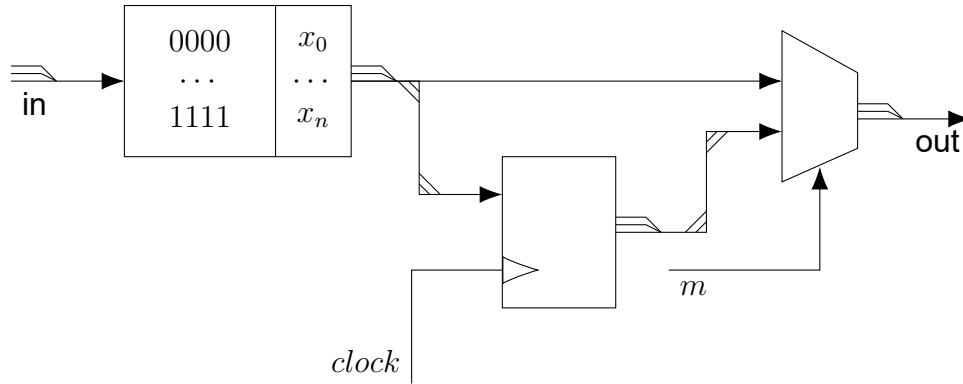


Figure 2.5: An n -input logic cell. x_k and m must be configured for any $0 \leq k \leq n$

FPGA manufacturers often group Logic cells in Configurable Logic Blocks (CLB) that make hardware production, placement, and routing (Section 2.2) easier.

The main building blocks of FPGAs are CLBs that are connected via routing fabric. These are responsible for most of the computation of FPGA programs. FPGAs can have additional modules such as RAM and Digital Signal Processors that optimize storage and specialized arithmetic, respectively. Each of these modules' functionality could also be performed by a collection of logic cells at the cost of performance. In this research, we will only consider CLBs.

2.1.4 Routing

A single logic cell can only perform simple programs such as logic gates. The FPGA needs to connect different logic cells to perform any kind of logic operation, dependent on what program it needs to execute. The way logic cells and other modules are connected on a physical FPGA in a specific configuration is called the **routing** of an FPGA. On the most basic level, FPGAs perform routing with configurable switches. These are tiny modules on an FPGA board that are connected with > 2 wires, and can be configured to block signals from specific wires; the other wires can then send- and receive signals.

2.1.5 Pins

To supply the FPGA with input and to allow it to provide output, an FPGA is equipped with a set of metal pins. Each of those pins is connected with a wire on the FPGA board. Each pin can be used for either input or output, depending on the configured program. For example, an FPGA used to control an electric stepper motor will receive input with the requested motor speed and direction and will output signals to specific circuits that need to be activated to obtain that speed and direction.

2.2 FPGA compilation

To program an FPGA, an engineer has to specify a hardware design that describes the semantics of the program. They write these hardware designs in a Hardware Description Language (HDL). Commonly used languages for this purpose are VHDL and Verilog. They specify on an abstract level what functionality the program should have, preferably in a modular structure to improve maintainability.

The software then needs to translate this abstract description to a **netlist**: a description of logic, registers, specialized components, and how they are connected. Depending on the configuration of the compilation software, this netlist may undergo a sequence of optimisations and transformations between several levels of abstraction. This process is called **synthesis**; beware though that some sources call the entire FPGA compilation process synthesis.

The next step in the compilation process is **placement**. The software maps each LUT, register, and other components to a physical place on the FPGA. The software can place components closer together to optimize the speed or can place components further apart to increase the probability routes can be found. Finding the optimal placement for speed is a proven NP-hard problem.

The last step in the compilation process is **routing**. State-of-the-art FPGA compilation pipelines perform this step sequentially after placement[28]. With the components locked in place, the software attempts to find a configuration of routing switches such that each connection that the synthesis requires is made. Finding the optimal routes for speed is also an NP-hard problem. However, finding any matching routing configuration is an NP problem.

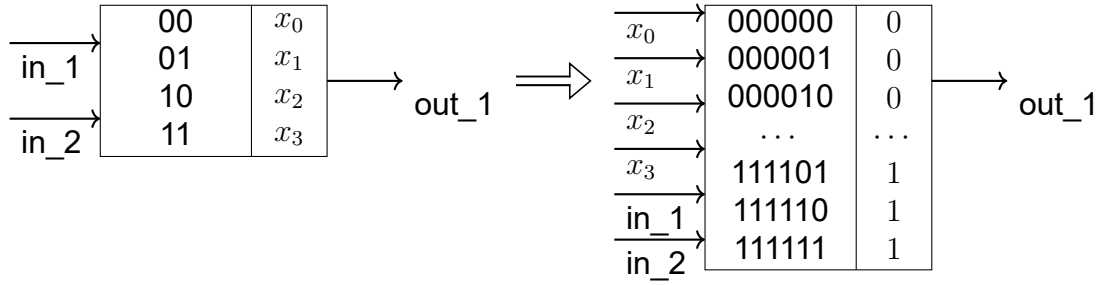


Figure 2.6: Simulation of an FPGA: the programming of the virtual FPGA is reflected in additional input of the actual FPGA. The value of x in the simulation is identical to the configuration of the program to be simulated.

! Note that place & route of an FPGA *program* as described here is a very similar problem to the placement of an unconfigured virtual FPGA. Place & route of an FPGA program has significant amounts of existing research, whilst placement of a virtual FPGA does not and is part of this research. The two problems are on different levels of abstraction: an FPGA program does not have the concept of unconfigured components or routing switches, while a virtual FPGA does.

Placement of FPGA programs using place & route could theoretically also work on a higher abstraction level, i.e. including unconfigured components. The problem with this approach is twofold: state-of-the art place & route pipelines perform these algorithms sequentially: they perform placement based on speed- and routeability heuristics without a guarantee that the placement can indeed be routed[28]. With the placement of FPGA programs, this is not a problem. Entire FPGAs, however, can have dense collections of components that can severely impact routeability. Secondly, it requires the virtual components to be compatible with the concrete components, which is not guaranteed.

Not using well-founded placement algorithms based on heuristics will undoubtedly have an adverse impact on the speed of our placement. However, we expect that we can use the hierarchical structure of an FPGA for significant improvements that could not be obtained with the placement of FPGA programs.

2.2.1 Literature

Yosys[51] is a free, open-source tool that performs general synthesis. These attributes make it highly usable in education. It can perform modular optimisations and transformations on netlists depending on a script, and outputs netlists in BLIF[10] format. It divides synthesis into three steps:

- Behavioural synthesis - converting an HDL program to Register Transfer Level (RTL): a netlist graph of high-level modules such as adders, multiplexers et cetera.

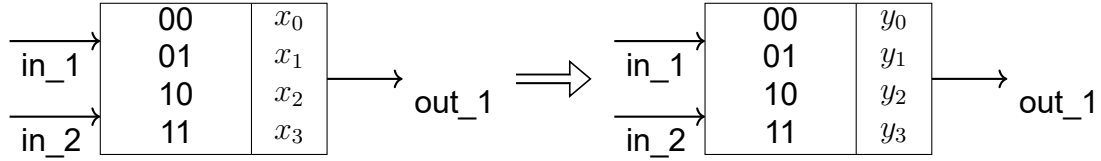


Figure 2.7: Emulation of an FPGA: the programming of the virtual FPGA is reflected in the programming of the actual FPGA. In this example, y_i takes the value of x_i . A less direct mapping could occur in emulation when, for example, a MUX is emulated by a LUT, or the inputs have a different permutation.

- RTL synthesis - converting an RTL netlist to a netlist graph of low-level logic gates and simple registers.
- Logic synthesis - converting a netlist graph of logic gates to a netlist graph of components available on specific hardware, such as LUTs and larger registers.

Much literature exists on FPGA synthesis. The proposed research resembles logic synthesis, but does not involve the application of synthesis techniques. These techniques rely on the absence of unconfigured components and an input that resembles an FPGA configuration, which we both do not have.

2.3 FPGA simulation

In this research, we will not be implementing FPGA simulation. However, for clarity, we will explain what we mean with simulation. Imagine a software engineer wants to execute a program P . Conventionally, they would program P in an HDL, compile it to an FPGA configuration file and load it onto the hardware.

To instead simulate the synthesized program, the engineer can also program a generic FPGA environment in an HDL, compile it to an FPGA configuration file and load it onto the hardware. They can afterwards provide P as *input* to the program instead of using P as a program. A simulation of a single LUT is shown in Figure 2.6.

Although engineers could use simulation to execute FPGA programs on FPGA hardware that the FPGA software was not compiled for, it introduces a considerable delay in execution. After all, many more components on the FPGA need to be used in the execution of the same program.

2.4 FPGA emulation

Instead, our research entails finding out how we can perform *emulation*. With this, we mean the execution of a program on a different FPGA *such that no extra input is required*. Instead, the emulator software inspects the program for the virtual FPGA and generates

what program on the concrete FPGA would have the same semantics given the same inputs as the source FPGA. Figure 2.7 shows the emulation counterpart of Figure 2.6. Since the target FPGA has the same LUT type in this example, the emulation converts the LUT configuration to an identical configuration on the target FPGA. We aim to find out how we can perform this kind of emulation with any program with any fixed source- and destination FPGA whenever such emulation is possible.

2.5 Graph theory

Because of the network structure of FPGAs, it is easy to model them as graphs. If we model FPGA components as combinations of vertices and connections, we have a complete representation of an FPGA suitable for graph algorithms. Let us suppose that we find a subgraph embedding of a graph representation of a virtual FPGA model in a graph representation of a concrete FPGA model. Then, by copying configurations from components in the virtual FPGA model to their concrete mapping we would have an emulation function!

The graph theoretic name for finding these embeddings is subgraph isomorphism. It is an NP-complete problem with many algorithms explored. The problem with using an approach of subgraph isomorphism is, however, that many possible emulations cannot be found. For example, a concrete FPGA may have much more configurable routing switches than a virtual FPGA. A subgraph isomorphism algorithm would in this case not find an emulation, even though one is technically possible by configuring some routing switches to function as wires.

A variant of subgraph isomorphism is subgraph homeomorphism. In this problem, intermediate vertices are allowed in the embedding on the target-graph side. When these intermediate vertices may be used exactly once (as in our use case), it is called node disjoint subgraph homeomorphism. This means that the embedding consists of a vertex-to-vertex mapping and an edge-to-path mapping.

Definition 2.5.1 (graph). A graph is a tuple (V, E, L) where V is a set of vertices, $E \subseteq (V \times V)$ is a set of undirected edges and $L \subseteq (V \times P(\lambda))$ is a multilabelling function where P is the powerset function and λ is a finite alphabet.

Definition 2.5.2 (vertex disjoint subgraph homeomorphism). Let P the set of all directed paths in G_2 , let $first(p)$ be the first vertex of a path p , let $last(p)$ be the last vertex of a path p and let $intermediate(p)$ be all other vertices of a path p . Then, a vertex disjoint subgraph homeomorphism from G_1 to G_2 is a tuple (f, q) where $f \subset (V_1 \times V_2)$ and $q \subset (E_1 \times P)$ are both injective relations such that:

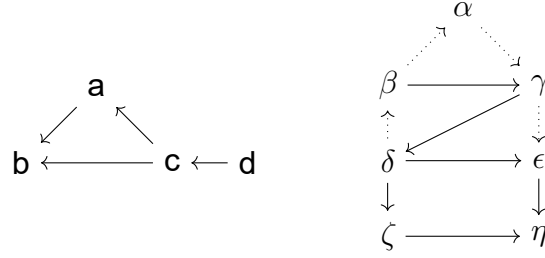


Figure 2.8: Two graphs G_1 and G_2 . G_1 is (node disjoint) subgraph homeomorphic to G_2 with the mapping $\{(a, \epsilon), (b, \eta), (c, \delta), (d, \beta), (a \rightarrow b, \epsilon \rightarrow \eta), (c \rightarrow a, \delta \rightarrow \epsilon), (c \rightarrow b, \delta \rightarrow \zeta \rightarrow \eta), (d \rightarrow c, \beta \rightarrow \gamma \rightarrow \delta)\}$. Other homeomorphisms exist as well.

1. $\forall u \in V_1. L_1(u) \subseteq L_2(f(u))$
i.e. vertices are mapped to vertices that have at least the same label set.
2. $\forall (u_1, u_2) \in E_1. \exists! p \in P. first(p) = f(u_1) \wedge last(p) = f(u_2) \wedge ((u_1, u_2), p) \in q$
i.e. each edge in G_1 is mapped to a path in G_2 .
3. $\forall p \in P. \forall x \in intermediate(p). ``arc'' \in L^V_2(x) \wedge$
i.e. every intermediate vertex in such a path is a routing switch, and...
 $\nexists p' \in (P \setminus \{p\}). x \in intermediate(p')$
...these routing switches may be used in no more than one path.

If we have graphs $G_{virtual}$ and $G_{concrete}$, then wherever a subgraph homeomorphism from $G_{virtual}$ to $G_{concrete}$ exists, it describes a mapping where the logic of the virtual FPGA may be performed on the concrete FPGA in an emulation²³. The set P of paths describe how nodes in the concrete FPGA are connected via paths of intermediate routing switches acting as wires.

The similarity between *subgraph isomorphism* and *subgraph homeomorphism* allows us to take inspiration from existing subgraph isomorphism algorithms and use similar methods to solve subgraph homeomorphism and therewith the FPGA emulation problem.

2.5.1 Literature

We performed literature research to obtain existing subgraph isomorphism and subgraph homeomorphism algorithms, the results of which are shown in Appendix .1. We searched the Scopus, ACM and IEEE libraries for the terms “subgraph isomorphism”, “subgraph matching”, “subgraph homeomorphism” and “topological minors”. We examined matching papers and selected those that solve exact (not approximate) subgraph isomorphism

²If no such relation exists, it does not mean a function f as specified in Section 3 does not exist. Emulation could still be obtained by emulation of components by (multiple) components of possibly different types or by emulation of routing switches by sets of connected routing switches.

³This does not hold if some vertices in $G_{concrete}$ that are part of the mapping are unconfigurably connected and their mapped equivalents in $G_{virtual}$ are not. This is, however, rare and we account for this in our algorithm.

or homeomorphism. We then iteratively added other algorithms from performance comparisons and citations. In the chart, an arrow implies that the paper of the newer algorithm or a survey paper showed that the algorithm at the source of the arrow performed better (i.e. lower mean time to find a matching) than the algorithm at the target of the arrow. A line between algorithms without arrowhead means a paper showed that the algorithms perform equally well.

Subgraph isomorphism

Malik et al.[39] simplify subgraph isomorphism by repeatedly randomly simplifying the target graph using a colouring.

PLGCoding[64], based on LSGCoding[63] uses the length of the shortest path and “Laplacian spectra” to effectively index the target graph and search for subgraphs. However, the invariants that these methods use to solve subgraph isomorphism do not necessarily hold for subgraph homeomorphism.

subISO[1] divides the target graph into subregions based on a pivot vertex in the pattern graph such that the number- and size of subregions is minimal. It then uses Ullmann[50] to search these subregions for the subgraph. Similarly, InfMatch[38] uses heuristics to select a node in the target graph and selects subregions in the target graph from that node to search in. PTSim[53] also applies graph partitioning, continuing with a different subgraph isomorphism algorithm on the resulting partitions, but only after first removing every edge from the target graph if the combination of labels of its source and target does not occur in the pattern graph. COSI[11] uses partitioning of graphs in cloud networks to find subgraph queries in social network graphs. Afterwards, it uses L2G’s algorithm[2].

GRASS[7], a subgraph isomorphism algorithm for GPUs, solves the problem by iteratively alternating between DFS and BFS of partial matchings on GPU cells as deep as the GPU architecture allows. Since the search space of subgraph homeomorphism is much larger than that of subgraph isomorphism, this may cause problems on GPU cells with limited memory resources.

VF2++[30] is an adaptation of VF2+[12], an improvement on VF2[17] which in its turn is an improvement of VF[16], a DFS algorithm in the search space of partial matchings in which the target graph is pruned using feasibility sets and the nodes in the pattern graph are matched in an efficient, fixed order. MuGram[34] is a variation upon VF2 that allows multiple labels on each node. Since VF2++ has not been compared with RI-DS[6], it is unknown whether a fixed node order is a more promising technique than a dynamic node order.

RI[8] is similarly to VF2++ a variant of DFS. It assigns a variable to each node in the pattern graph with the domain of possible matches in the target graph. In its DFS, it uses a *dynamic* ordering of nodes. A node in the target graph is matched next if it has many neighbours in the existing partial matching, many neighbours of neighbours in the existing

partial matching or else a large degree. With this, RI aims to maximize the number of verifiable constraints. RI-DS[6] improves upon this by implementing a cache that checks label compatibility for nodes quicker. Rudolf[45] proposes a querying method to access graph data in a subgraph isomorphism problem within a constraint satisfaction problem context but does not provide an algorithm.

Similarly to RI, LAD[48] solves subgraph isomorphism using constraints. It then applies AllDifferent filtering (using an algorithm by Régin[43]) during constraint solving/DFS to reduce the domain as much as possible. This filtering removes every u from the domain of v whenever an assignment of v to u would result in an empty domain for some variable. McGregor[41] performs this as well but with a different AllDifferent algorithm. PathLAD[33] improves upon LAD by checking each match whether the number of 3-cycles of connected nodes in the target graph is at least as high as the number of 3-cycles in the pattern graph where it is matched.

CLF-Match[5] speeds up subgraph isomorphism by splitting the pattern graph into a dense core of well-connected nodes and sparse trees attached to it. By matching the core first, it avoids many unfruitful partial matches in DFS. Furthermore, it introduces a CPI data structure. This data structure helps to find subgraph isomorphisms more efficiently and takes polynomial time to construct.

LLAMA[33] and BM1[3] are portfolio techniques. Based on heuristics, they pick an approach from a collection that they expect to perform best. LLAMA picks from a collection of different algorithms whilst BM1 picks from different pruning settings for VF2.

BoostISO[44] introduces a filtering optimization for DFS: whenever a node u in the pattern graph matches with a node v in the target graph and it fails, any v' in the target graph with a subset of the neighbours of v will be disregarded as candidate match for u . It furthermore introduces a data structure that finds many subgraph isomorphisms as soon as one subgraph isomorphism is found.

Glassgow[40] is a DFS algorithm with dynamic node ordering. Other than other often used filtering techniques to disregard potential matches, it introduces backjumping: whenever a finished recursive call fails to match a node v in the source graph, it jumps back to the last time the domain of v was changed. Furthermore, it introduces supplemental graphs: whenever a filter removes matches from the domain of a supplemental graph, it may also be removed from the domain of the original graph.

TurboISO[25] extracts subregions from the target graph by finding instances of a compressed tree (NEC) of the pattern graph in the target graph (a polynomial process). Furthermore, it proposes using the results of this DFS to generate a vertex order to be used in BFS for subgraph isomorphism.

SQBC[61] takes cliques into account when searching for subgraphs: any node in a maximal clique in the pattern graph needs to be matched with a node in a maximal clique in the target graph that is at least as large.

STW[49] splits the pattern graph into small pieces and attempts to find all occurrences of a graph piece. Within this set, it then iteratively searches for the next piece.

GraphQL[27] filters the domain of source graph nodes based on the fact that neighbourhoods of nodes need to be sub-isomorphic to matched nodes in the target graph. It furthermore filters based on bijections from- and to adjacency subtrees. ILF [56] formalizes this sub-isomorphism by iteratively strengthens the filtering power of labels until a fixed point indicates sub-isomorphisms. It uses these labels to reduce the domains of each source node and then updates labels to be as strong as possible.

NOVA[62] computes for each node v in the source vertex a *signature*: for each node u of distance $x < c$ it lists its label and the number of paths from u to v of length x . It uses this signature to filter out false matches from the domain of source graph nodes.

Treepi[57] and Gaddi[58] use the distances between node pairs to index graphs. Sing[18] improves upon this by indexing the graph on the fly during the search process. SPath[60] also uses indexing techniques on trees and subgraphs to speed up the search for a complete mapping.

Subsea[37] recursively cuts the target graph along its minimal cut, searches for the subgraph on the edges on this cut and then continues in the resulting two subregions. This algorithm assumes that a minimal cut has few edges and that the target graph is much larger than the source graph.

QuickSI[47] makes use of a set of spanning entries to combine tree search with normal DFS.

Cheng[13] proposes a method of storing constraints as matrices and performing matrix operations on Ullmann's[50] representation of partial mappings.

Ullmann[50] is often used as a baseline algorithm when testing subgraph isomorphism algorithms. It performs DFS using nodes with the highest degree in the source graph with random nodes in the target graph. L2G[2] improves upon this by selecting unassigned nodes from the target graph that are connected to the partial matching first. Fast-ON[23] improves upon Ullmann's algorithm by ordering the pattern graph vertices by degree and taking labels into account. UI[14] improves upon Ullmann's algorithm[50] by ordering the vertices of the pattern graph by a 'subdegree' measure in descending order and by breaking ties by choosing the node with the highest closed-neighbourhood clustering value.

From this literature research, we conclude that a DFS for partial mappings is a viable approach to subgraph isomorphism and thus potential to subgraph homeomorphism. Many algorithms, however, use incompatible strategies to obtain subgraph isomorphisms. Experimentation will have to show what strategy is effective for path subgraph isomorphism.

Subgraph Homeomorphism

Lingas et al.[36] present an algorithm for subgraph homeomorphism under the assumption the vertex placement is fixed. For general subgraph homeomorphism they suggest trying their algorithm on each possible vertex matching.

Xiao et al.[52] present an algorithm for subgraph homeomorphism when the length of the intermediate paths is in a limited range⁴. Since our path lengths can be $[1, -\infty)$, we would need to find an alternate solution.

Grohe et al.[24] show that for every fixed source graph, an $O(|V_{target}|^3)$ algorithm exists that can find homeomorphic embeddings in any target graph. However, finding this algorithm is a non-trivial, NP-hard process.

LaPaugh et al.[35] present some ways to reduce the graphs in a subgraph homeomorphism problem. These reductions are, however, not applicable to FPGA graphs.

⁴The algorithm involves precomputing all possible paths. The number of possible paths increases exponentially with the sizes of both the pattern- and target graph.

3 OBJECTIVES

This research entails the discovery of how to emulate a virtual FPGA on a concrete FPGA. To this purpose, we have established the following research question:

Given...

- a graph specification of the structural layout of a virtual FPGA A ,
- a graph specification of the structural layout of a concrete FPGA B ,

how do you assemble a linear¹ function f such that for any representative model of a program x for model A , $f(x)$ is a representative model of a program for model B that is semantically equivalent?

Subquestion 1

How do the computational- and space requirements of the generation (not execution) of f scale with the size of FPGAs A and B ?

Subquestion 2

How much wiring and how many components does the concrete FPGA need for emulation of a virtual FPGA of complexity x ? Is this practical for use in education?

Expected outcomes

We expect an algorithm and an implementation of that algorithm that, given two FPGA models, generates a function that translates a model of an FPGA program with compatibility with one FPGA model to a model of an FPGA program with compatibility with the other. Furthermore, we include specifications on how to model FPGA models for this purpose.

¹i.e. a numeric constant c exist such that the number of instructions required in the execution of f is less than $c * (\text{the number of components of the virtual FPGA})$

4 MODELS

Our algorithm will generate an emulation using vertex disjoint subgraph homeomorphism. For this purpose, we will use vertex-multilabeled directed simple graphs as defined in Definition 2.5.1. In these graphs we will:

- ..model a wire as a vertex that may be connected with any number of vertices in either direction of types “*arc*”, any number of incoming edges from “*port_out*” vertices and any number of outgoing edges to “*port_in*” edges.
- ..model a transistor as a vertex with a label “*arc*” and a label “*configurable*” for configurable transistors or label “*unconfigurable*” for unconfigurable transistors (that are not counted towards label compatibility). It has an incoming edge from the source “*wire*” of the transistor and an outgoing edge to the target “*wire*” of the transistor.
- ..model a LUT as a vertex with label “*lut*” that only has connections from vertices with label “*port_in*” and to vertices with label “*port_out*”.
- ..model a register as a vertex with label “*register*” that only has connections from vertices with label “*port_in*” and to vertices with label “*port_out*”.
- ..model an input of a component as a vertex with label “*port_in*” and an outgoing directed edge towards a “*lut*” or “*register*” vertex and an incoming edge from a “*wire*” vertex.
- ..model an output of a component as a vertex with label “*port_out*” and an incoming directed edge from a “*lut*” or “*register*” vertex and an outgoing edge to a “*wire*” vertex.

An example of such a model is shown in Figure 4.1 where a graph model is shown of the logic cell in Figure 2.5.

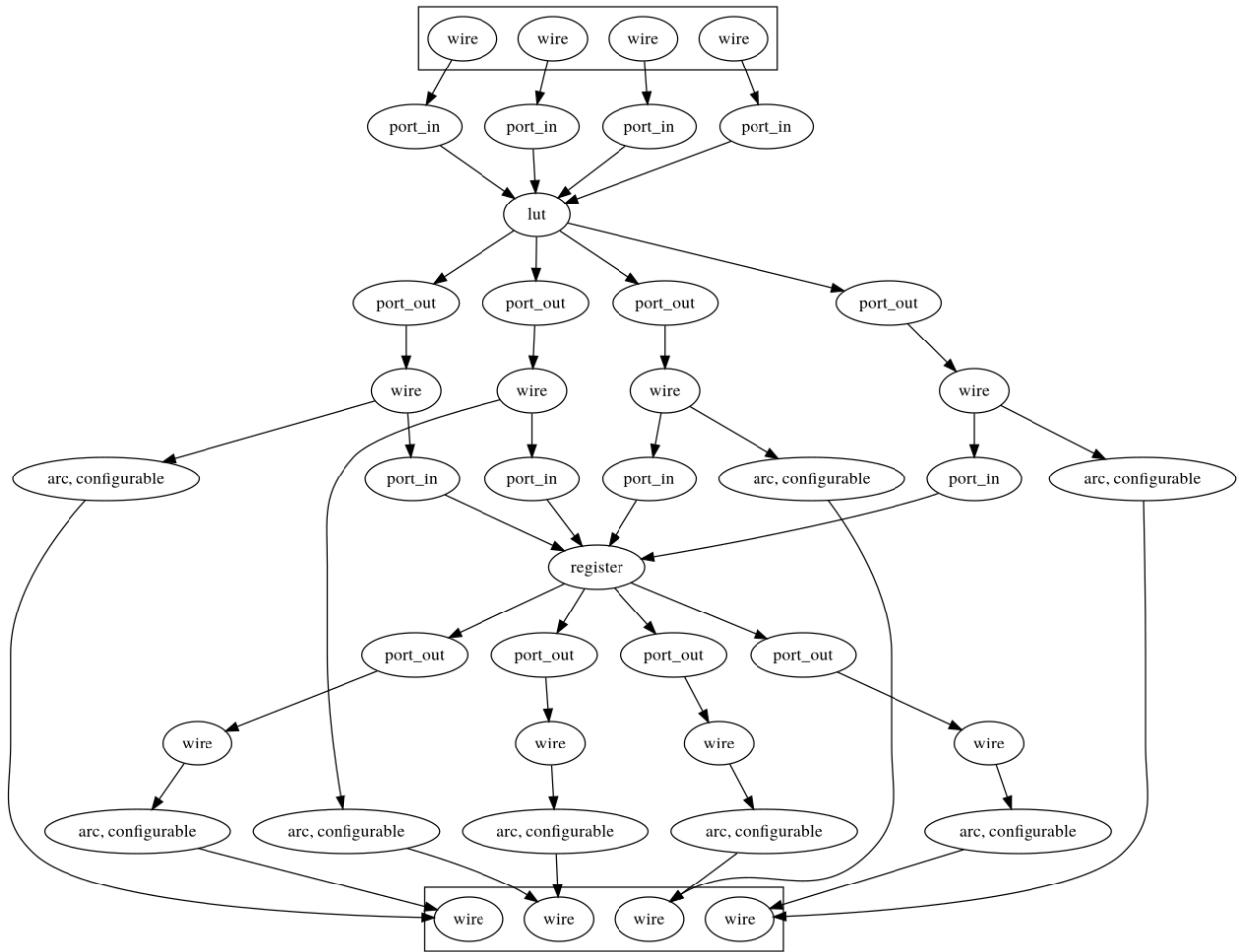


Figure 4.1: Graph model of the logic cell shown in Figure 2.5

5 ALGORITHM

Algorithm 1: ndSHD2

Inputs: the current matching state s , the current node compatibility matrix M , the current independent path matrix R

Outputs: *found*, indicating whether a valid homeomorphism has been found.

if s is dead state **then**

return false;

end

else if s is complete **then**

return true;

end

$found \leftarrow false$

initialization;

while $\neg found \wedge \exists$ valid node/edge-path mapping pair **do**

$m \leftarrow GetNextNodePair() // m \leftarrow GetNextEdgePathPair();$

$s' \leftarrow BackupState(s);$

$NodeMap_s \leftarrow NodeMap_s \cup \{m\} // EdgePathMap_s \leftarrow EdgePathMap_s \cup \{m\};$

$Refine(M, R);$

$found \leftarrow ndSHD2(s, M, R);$

if *found* **then**

return true;

end

$s \leftarrow RecoverState(s');$

end

return false;

5.1 Baseline: ndSHD2

In our research we will extend existing work on node disjoint subgraph isomorphism. In the literature we found two existing concrete algorithms. We choose to adapt Xiao's algorithm ndSHD2[52] as baseline over Lingas [36] because it includes methods for vertex-on-vertex

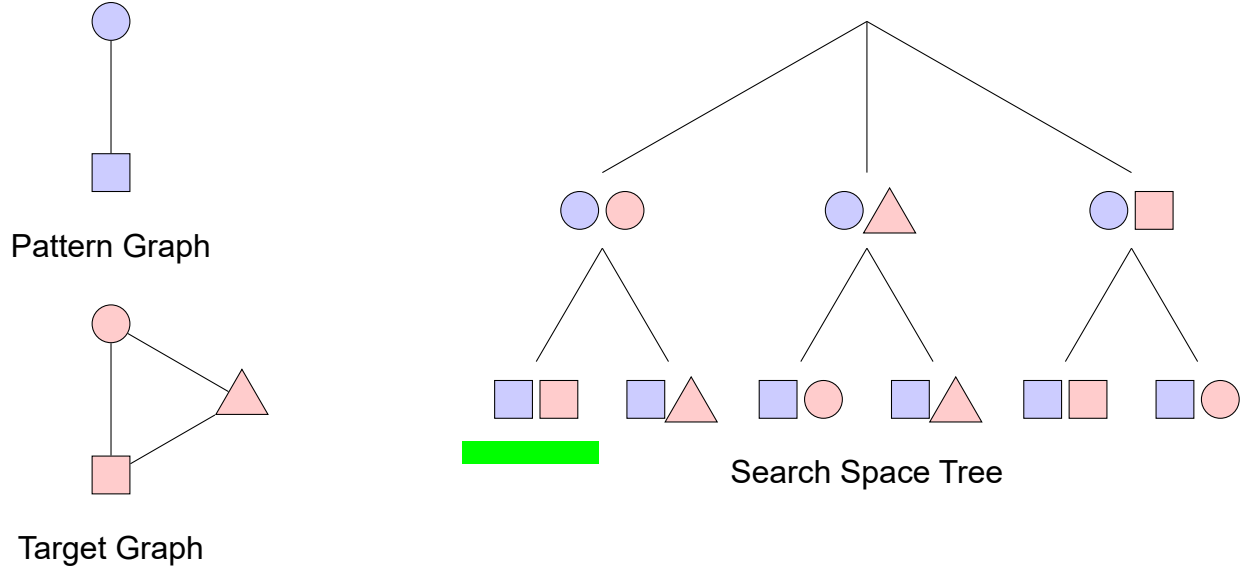


Figure 5.1: State space of subgraph isomorphism search algorithms. Vertices with the same shape may be matched to each other, while vertices with different shapes have incompatible labels. The green bar denotes the only search path that leads to a successful subgraph isomorphism. Source: [8]

mapping. This saves valuable time research vertex-on-vertex mapping strategies. We will adapt this algorithm to compute paths on-the-fly instead of beforehand to avoid the exponential space requirement.

ndSHD Is shown in Algorithm 1. It is a form of depth first search in a partial mapping search space that attempts- and backtracks vertex-on-vertex mappings and edge-on-path mappings. An example of such a search space is shown in Figure 5.1. Whenever a partial match is extended with a vertex-on-vertex mapping or with an edge-on-path mapping the search space is refined using a pruning strategy implemented by the *Refine*(M, R) call. This method filters out future path mappings that have an already mapped target graph vertex as intermediate vertex.

In our literature study, we found that the core of many subgraph isomorphism algorithms¹ is a form of DFS state space exploration as well where the states consist of partial vertex-to-vertex mappings. Xiao’s algorithm’s search is extended with a partial mapping from E_{source} to the path set of G_{target} .

The algorithm leaves some parameters open that affect the order in which search states are visited:

- The order in which source- **and** target vertices are retrieved with *GetNextNodePair*(),

¹Ullman, VF, VF2, VF2+, VF2++, UI, Fast-ON, L2G, Cheng, QuickSI, GraphQL, ILF, TurboISO, Glassgow, CLF-Match, RI, RI-DS, McGregor, LAD, PathLAD

- when *GetNextNodePair()* should be called and when *GetNextEdgePathPair()* should be called and
- the order in which edges- **and** paths are retrieved with *GetNextEdgePathPair()*.

We fill in these parameters based on observations from subgraph isomorphism algorithms.

5.2 Parameters

When to map vertices and when to map edges / ordering of edges

The paper does conclude that the performance was better when edges were mapped as early in the process as possible (immediately after both vertices of the edge have been mapped). We will use this strategy as well. In case of ties we choose the edge that connects with the earliest matched vertex in the search process.

The ordering of source graph vertices

To get a viable vertex ordering for subgraph homeomorphism we may use the most optimal vertex algorithm from subgraph isomorphism. From performance comparisons we extract that the best performing algorithm that adheres to partial mapping search for subgraph isomorphism is RI-DS. In our literature study, we found no evidence a faster algorithm exists.

To describe the ordering process of RI-DS, let us provide a few definitions:

Definition 5.2.1 (predecessors and successors). Given two graphs $G = (V, E, L)$ and some vertex $v \in V$, their predecessors and successors are defined as follows:

$$succ(v) := v' \in V. (v, v') \in E$$

$$pred(v) := v' \in V. (v', v) \in E$$

Definition 5.2.2 (E for vertices). If some $v \in V$, then $E(v) := succ(v) \cup pred(v)$.

Definition 5.2.3 (E for sets of vertices). If some $S \subseteq V$, then $E(S) := \bigcup_{v \in S} E(v)$.

The algorithm RI-DS obtains a variable ordering using a greedy algorithm called “GreatestConstrainedFirst”. This algorithm starts with an ordering μ containing only the source graph vertex with the highest degree, i.e. $v \in V_1$ where $|E(v)|$ is highest. Iteratively, it chooses the next vertex to be explored in partial mapping search. It selects each vertex v of which $|E(v) \cap \mu|$ is greatest. If multiple vertices have the greatest value, ties are broken by choosing the vertex for which $|E(E(v) \setminus \mu) \cap \mu|$ is greatest. If multiple vertices remain that have the same value, ties are broken by choosing the vertex for which $|E(E(v) \setminus \mu) \setminus \mu|$ is greatest. Any remaining ties are broken randomly.

The result of this ordering is that consequent vertices have many edges with vertices earlier in the ordering. Xiao established that matching edges earlier results in a faster algorithm. Using an order that allows early placement of edges such as GreatestConstrainedFirst should according to Xiao result in fast execution.

The ordering of target graph vertices

We found one subgraph isomorphism algorithm that uses a target graph vertex order that is *not* arbitrary, being Glasgow[40]. In this algorithm, target graph vertices with higher degree are prioritized. We will use the same target graph vertex ordering.

The ordering of target graph paths

In subgraph isomorphism, edge-on-edge mappings are trivial from vertex-on-vertex mappings. However, in subgraph homeomorphism a source graph edge may be mapped on many target graph path candidates (all starting- and ending at the same two vertices). Therefore, we cannot extract the order in which to try out paths from subgraph isomorphism. Instead, we implemented different methods to iterate over paths to try:

- **K-path** - Try all paths from shortest to longest, avoiding unusable vertices in the existing partial mapping.
- **DFS** - Search for paths using depth first search from the start vertex, choosing arbitrary directions at each vertex and avoiding unusable vertices in the existing partial mapping.
- **Greedy DFS** - Search for paths using greedy depth first search, choosing the direction closest to the goal vertex first and avoiding unusable vertices in the existing partial mapping.
- **Control point** - Select increasingly many 'control points' (from 0 to $|V|$) randomly in the target graph that must be in the path, then connecting them by a shortest path algorithm that avoids unusable vertices in the existing partial mapping.²

These methods provide for each two vertices in a graph each path connecting them exactly once. The space requirements for each method are shown in Table 5.1.

5.3 Optimisations

In addition to fixating some parameters of the ndSHD2 algorithm, we implement some optimisations (some of which from the domain of subgraph isomorphism) and individually evaluate them.

²To avoid duplicate paths, any path that could have been generated with fewer control points is skipped. Furthermore, any control point configuration where shifting some control point towards the goal vertex results in the same path is skipped.

Path iterator	Space complexity
K-Path	$O(V ^2)$
DFS	$O(V)$
Greedy DFS	$O(V ^2)$
Control point	$O(V)$

Table 5.1: Worst-case space complexity of each path iteration strategy

5.3.1 Refusing long paths

Since path iterators may provide any valid path to map an edge to during the matching process, they may also provide paths that take up unnecessarily many resources. Specifically, they take up so many resources that with a subset of the vertices and edges of that path, a shorter path can be formed. Formally, some path p is “unnecessarily long” iff:

$$\exists p' \in P. first(p') = first(p) \wedge last(p') = last(p) \wedge intermediate(p') \subset intermediate(p)$$

With this optimisation enabled, such paths are skipped by path iterators.

5.3.2 Runtime Pruning

Some subgraph isomorphism algorithms[17, 40] prune the search space during the search using some detection method of dead search paths. In Chapter 6 we will elaborate on the different techniques we implement for subgraph homeomorphism.

6 PRUNING

During the search for a complete matching, during which the algorithm only has *partial* matchings, the algorithm will often explore branches of the search tree that will not eventually lead to a homeomorphism. We will call these branches *dead branches*. Exploring an entire dead branch may be costly: its size is exponential¹ thus exploration costs an exponential amount of time. A solution to this is to implement methods of early detection of such dead branches, i.e. pruning methods. These methods can by nature not detect every dead branch efficiently². Therefore, there exists a tradeoff between strength, i.e. how many dead branches it can detect, and performance, i.e. how the number of instructions used by pruning scales with the size of the inputs.

Algorithms for subgraph *isomorphism* use various pruning methods, which we will implement for subgraph *homeomorphism*. Subgraph isomorphism is, however, a stricter problem than subgraph homeomorphism. Each subgraph isomorphism mapping is a node disjoint subgraph homeomorphism but not every subgraph homeomorphism is a subgraph isomorphism. Because of this, its pruning methods can filter away some search tree branches that could never result in a subgraph isomorphism, but that could possibly result in a node disjoint subgraph homeomorphism. This means that possibly fewer dead branches can be pruned in the search for subgraph homeomorphisms compared to subgraph isomorphism. In this research, we will experiment with different pruning methods applied to searching subgraph homeomorphisms and draw conclusions about their applicability.

The input of these pruning methods comes down to an assignment of domains (sets of target graph vertices) to variables (source graph vertices). These domains represent the target graph vertex candidates for each source graph vertex in vertex-on-vertex matching. Source graph vertices that are already in the partial matching have a domain of size one: the target graph vertex they have been matched to. The other vertices have domains that can be calculated in different ways (see Section 6.1). The pruning method then decides whether the algorithm should continue the search in this branch (i.e. do not prune) or whether the algorithm should backtrack. The different methods of deciding this are elaborated in Section 6.2.

¹due to the NP-completeness of node disjoint subgraph homeomorphism

²again, because of the NP-completeness of node disjoint subgraph homeomorphism.

6.1 Domain filtering

The goal of domain filtering during a search is to assign a domain of target graph vertices to each source graph vertex such that the following three criteria are satisfied:

1. If at least one homeomorphism can be found from this search branch, then for some homeomorphism that can be found from this search branch with vertex-on-vertex matching M_V it holds that $\forall (s \rightarrow t) \in M_V. t \in \text{domain}(s)$.
2. Each domain assignment contains as few ‘false positives’ as possible, where a false positive is a pair of a source vertex s and a target vertex t such that t is in the domain of variable s and no homeomorphism exists from the current search path in which s is matched to t .
3. The process of domain filtering has a computational complexity that is as low as possible.

In this section, we will explore different methods to obtain these domains, each with different tradeoffs between strength (performing better at criterium 2) and performance (performing better at criterium 3).

6.1.1 Labels and neighbours

The weakest and fastest method to obtain domains for some source graph vertex u is by selecting each unmatched target graph vertex v that has compatible labels and has compatible in- and outdegrees:

Definition 6.1.1 (compability under label constraint). If M is the current partial matching, then:

$$\text{compatible}_{\text{LABEL}}(u, v) := v \notin M \wedge L(u) \subseteq L(v) \wedge |\text{pred}(v)| \geq |\text{pred}(u)| \wedge |\text{succ}(v)| \geq |\text{succ}(u)|$$

This method satisfies criterium since every homeomorphism needs each source graph vertex u to be matched with a target graph vertex v that has at least the same label set as u , and the possibility of connecting with each mapped predecessor and successor of u . It has a low computational complexity per source-target pair ($O(|L_S|)$).

6.1.2 Free neighbours

A somewhat stronger and somewhat slower method to obtain domains in addition to label filtering is to compare the indegree- and outdegree of each unmatched source graph vertex u to other unmatched source graph vertices to the in- and outdegree of the target graph vertex v to other unmatched target graph vertices. The target graph vertex must have a larger or equal indegree and outdegree compared to the source graph vertex:

Definition 6.1.2 (compatibility under free neighbours constraint). If M is the current partial matching, then:

$$\text{compatible}_{FN}(v, u) := \text{compatible}_{LABEL}(v, u) \wedge C_{indegree}(v, u) \wedge C_{outdegree}(v, u)$$

where:

$$C_{indegree}(u, v) := |\{v' \in \text{pred}(v).v' \notin M\}| \geq |\{u' \in \text{pred}(u).u' \notin M\}|$$

$$C_{outdegree}(u, v) := |\{v' \in \text{succ}(v).v' \notin M\}| \geq |\{u' \in \text{succ}(u).u' \notin M\}|$$

This method satisfies criterium since

6.1.3 Reachability of matched vertices

A strong (and computationally very expensive) method of filtering the domain we will use³ for some source graph vertex u and some target graph vertex v in addition to label- in-degree and outdegree compability is to check that v can reach the mapped vertices of successors of u and that v can be reached from the mapped vertices of predecessors of u , i.e. that candidate paths exist for the upcoming edge-path matching attempts. This pruning method does not check whether a set of paths exist that is vertex disjoint since that is the task of the main algorithm: it merely checks the existence of paths. If no path exists, then no vertex disjoint path exists and pruning is required. An example of a domain filtered by reachability can be found in Figure 6.1. Formally:

Definition 6.1.3 (compatibility under reachability constraint). If M is the current partial mapping and P is the set of paths in the target graph, then:

$$\text{compatible}_{REACH}(v, u) := \text{compatible}_{FN}(v, u) \wedge C_{inr}(v, u) \wedge C_{outr}(v, u)$$

where:

$$\begin{aligned} C_{inr}(u, v) &:= \forall u' \in (\text{pred}(u) \cap M). \exists p \in P. \begin{aligned} &\text{first}(p) = M(u') \quad \wedge \\ &\text{last}(p) = v \quad \wedge \\ &\nexists i \in \text{intermediate}(p). i \in M \end{aligned} \\ C_{outr}(u, v) &:= \forall u' \in (\text{succ}(u) \cap M). \exists p \in P. \begin{aligned} &\text{first}(p) = v \quad \wedge \\ &\text{last}(p) = M(u') \quad \wedge \\ &\nexists i \in \text{intermediate}(p). i \in M \end{aligned} \end{aligned}$$

6.1.4 Reachability of neighbourhood

In addition to filtering domains based on the current partial matching and the source- and target graph, we can perform domain filtering based on domains that are already computed. If some source graph vertex s_1 with domain D_1 has an edge to some vertex s_2 with domain D_2 , then for each target graph vertex $t_1 \in D_1$ there must exist a path to some vertex $t_2 \in D_2$. If not, then t_1 may be removed from D_1 . We will call this filtering *level 1 neighbourhood filtering*. If we D_2 was itself filtered using level 1 neighbourhood filtering, we call this *level 2 neighbourhood filtering*. Similarly, each additional time the neighbourhood

³An even stronger method would be to find paths that each have different second vertices and each have different second-last vertices, or even to check that no intermediate vertex is used multiple times (NP-complete). This is, however, out of scope for this research.

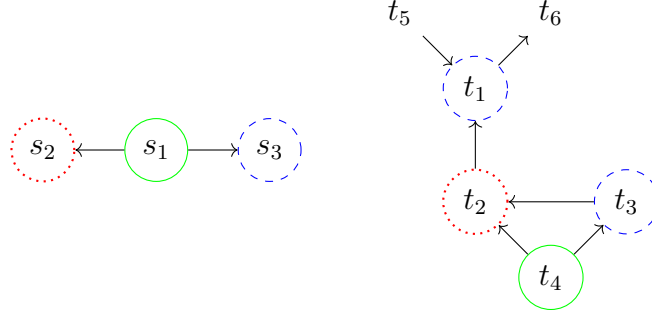


Figure 6.1: After vertex placements $s_1 \rightarrow t_4$ and $s_2 \rightarrow t_2$, the domain for vertex s_3 would normally be $\{t_1, t_3\}$. However, by checking for reachability from t_4 we can reduce this to $\{t_3\}$. The numbers represent the matching order and the circle styles represent labels.

Source graph vertex	Target graph candidates
s_1	$\{t_1, t_3, t_5\}$
s_2	$\{t_1, t_2, t_3, t_4\}$
s_3	$\{t_2, t_3, t_6\}$
s_4	\emptyset

Table 6.1: If the empty domain pruning method detects an empty target graph domain for some source graph vertex, backtracking is initiated. This is the case if the possible target graph candidates are as shown as in this table.

reachability filtered domain is reused in another neighbourhood reachability filtering we consider it of a higher level. When computing domains serially, an appropriate maximum level should be used: using level 2 neighbourhood filtering over level 1 neighbourhood filtering yields, for example, much more domain reduction and is much less computationally demanding compared to level 20 neighbourhood filtering over level 19 neighbourhood filtering. However, if a cached approach is used (see section TODO), we can perform level ∞ neighbourhood filtering using a fixed point algorithm.

6.2 Pruning methods

6.2.1 ZeroDomain pruning

The ZeroDomain pruning method decides to backtrack if and only if one of the domains is empty, i.e. there exists some source graph vertex that cannot be matched with any target graph vertex. A homeomorphism cannot be found in the current search branch as no potential match exists for this source graph vertex. An example of a domain assignment where ZeroDomain prunes the search space is shown in Table 6.1.

Source graph vertex	Target graph candidates
s_1	$\{t_1, t_2, t_3\}$
s_2	$\{t_1, t_2\}$
s_3	$\{t_2, t_3\}$
s_4	$\{t_1, t_3\}$

Table 6.2: In this example, four source graph vertices have a total domain of only three target graph candidates. By the pigeonhole principle, no injective assignment is possible. AllDifferent recognises this and initiates backtracking.

6.2.2 AllDifferent pruning

The AllDifferent constraint specifies that every variable should be able to have a different value. This is the case with some variable having an empty domain (i.e. AllDifferent is stronger than ZeroDomain), but also in other cases. Since a homeomorphism requires such an assignment, the AllDifferent pruning algorithm backtracks in this case. Unfortunately, we could not find an AllDifferent implementation using less than quadratic space. An example of a domain assignment where AllDifferent prunes the search space but ZeroDomain does not is shown in Table 6.2.

6.3 When to apply

When a pruning method has been selected and a strategy to obtain the domains, one must lastly decide when to apply the pruning method.

6.3.1 Runtime calculation

The simplest option is to calculate the domains and run the pruning strategy each time a vertex is selected for usage in a matching. This could be a target graph vertex that is selected as for vertex-on-vertex matching or a target graph vertex that is part of a path in edge-on-path matching. This changes the partial matching and thus the domains. The pruning method then decides to allow it or to disallow it.

6.3.2 Caching domains - incremental domain calculation

Another option is to cache the domain of each variable and update it based on the current partial matching. This saves valuable time but requires quadratic space⁴ (for each source graph vertex $\in |V_{source}|$ it needs to store a domain of average size $O(|V_{target}|)$). If the reachability pruning method is used, paths need to be cached as well. That is, for each edge $\in E_{source}$ we need to store a path of $O(|V_{target}|)$ vertices.

⁴i.e. some constant c exists such that the space required has an upper bound of $c * |V_{target}|^2$

6.3.3 Parallel calculation

Finally, we can perform the domain filtering and procedure in a separate CPU thread while the algorithm continues without backtracking. The pruning thread queries the current matching and calculates whether pruning is appropriate for that matching. If it decides that it is not, it requeries the current matching. Otherwise, it will interrupt the main thread and signal it to backtrack until the pruning method does not detect a dead branch anymore. This method cannot perform caching

7 USE CASE: LATTIC ECP5

We will apply our algorithm to map virtual FPGAs to the concrete Lattice ECP5 FPGA. An image of this FPGA on an evaluation board is shown in Figure 7.2. This FPGA's architecture consists of 'tiles' of different types in a grid pattern. Each of these tiles has an associated x- and y-coordinate in the grid system and is (generally) topologically the same as tiles of the type elsewhere in the grid. There exists I/O tiles which function is to retrieve and send data from outside the FPGA, DSP tiles that perform signal processing calculations, tiles that only provide routing structure, RAM tiles and tiles that are internally used for clock management and configuration.

The structure of an ECP5 FPGA is shown in Figure 7.3. This figure shows the grid/tile structure of the ECP5 along with specific components from the electrical engineering domain. The largest portion of the grid structure are logic tiles that contain 8 LUTs and 8 registers (or flip-flops (FF)). We will focus on this type of tile to find homeomorphisms.

The logic cell of

The virtual FPGA we aim to emulate on this board (which we will call *VirBoard*) is one that, like the ECP5 consists of a tile-like structure with no wires spanning no more than a single tile. Because of this limitation, a student may freely drag-and-drop functionality in the virtual environment without worrying about implicitly overlapping connections. Each tile in this virtual FPGA has significantly less functionality than one of the ECP5. It has a single in- and output at each wind direction that may be connected in various ways, a 2-bit LUT and an 1-bit register. A graph model of VirBoard is shown in Figure 7.1.

¹<http://www.latticesemi.com/products/developmentboardsandkits/ecp5evaluationboard>, accessed July 10th 2020

²ECP5 and ECP5-5G Family Data Sheet FPGA-DS-02012-2.1 (April 2019), Figure 2.1. Original caption: Simplified Block Diagram, LFE5UM/LFE5UM5G-85 Device (Top Level)

³ECP5 and ECP5-5G Family Data Sheet FPGA-DS-02012-2.1 (April 2019), Figure 2.2. Original caption: PFU Diagram

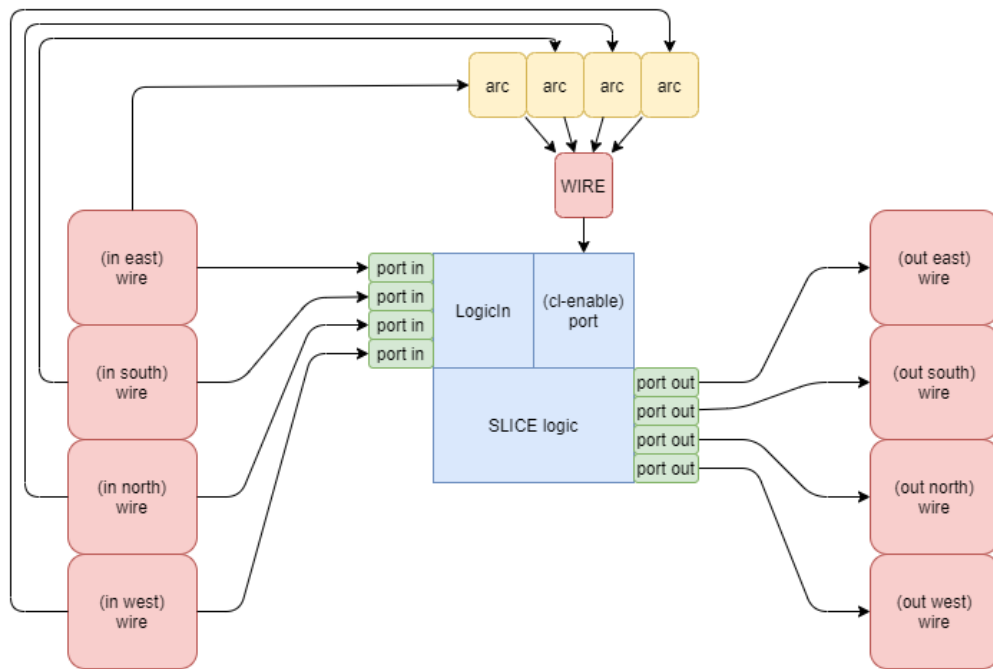


Figure 7.1: A graph model of a single cell of the virtual FPGA aimed to emulate on the Lattice ECP5

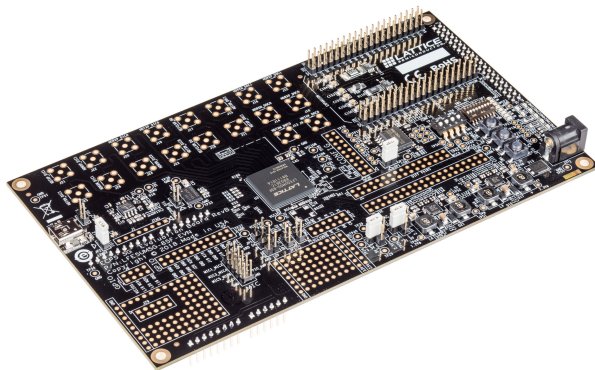
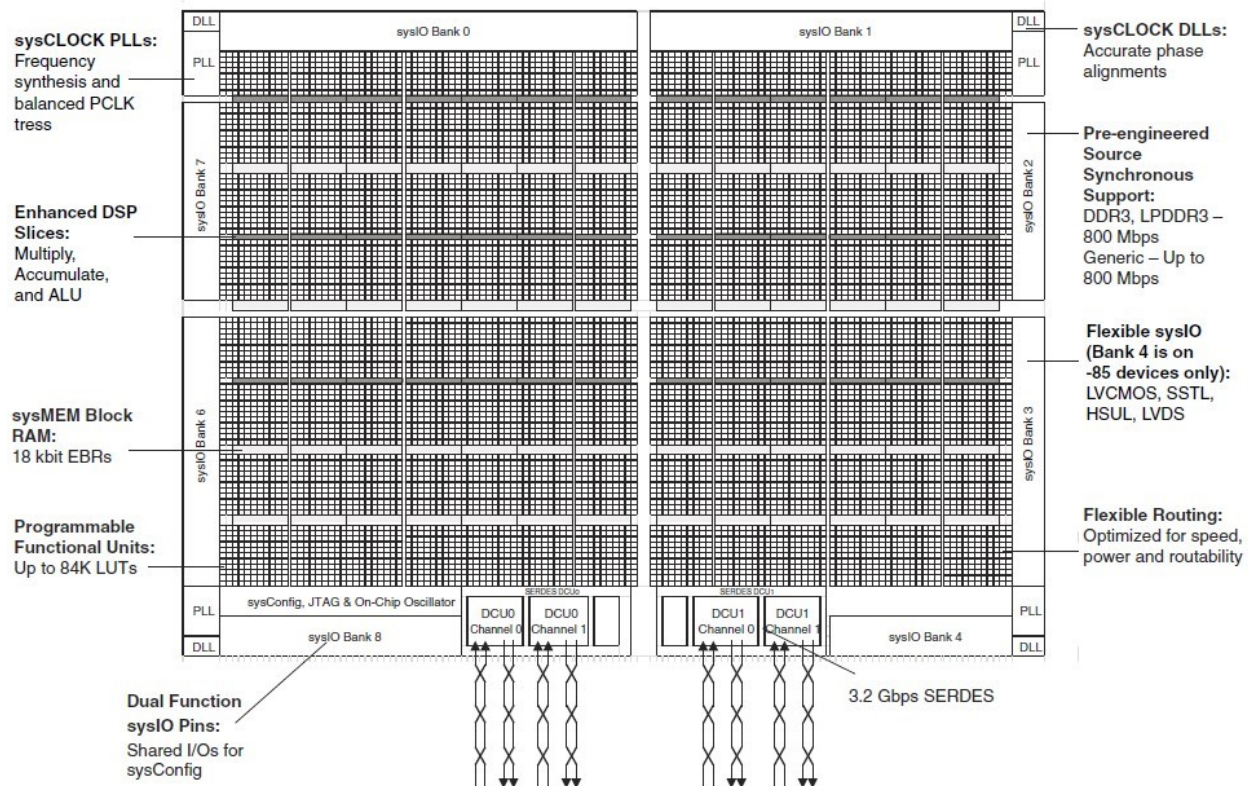


Figure 7.2: Evaluation board of the LFE5UM5G-85F FPGA: a variant of the ECP5.¹



Note: There is no Bank 4 in -25 and -45 devices.
There are no PLL and DLL on the top corners in -25 devices.

Figure 7.3: The architecture of a Lattice ECP5 FPGA².

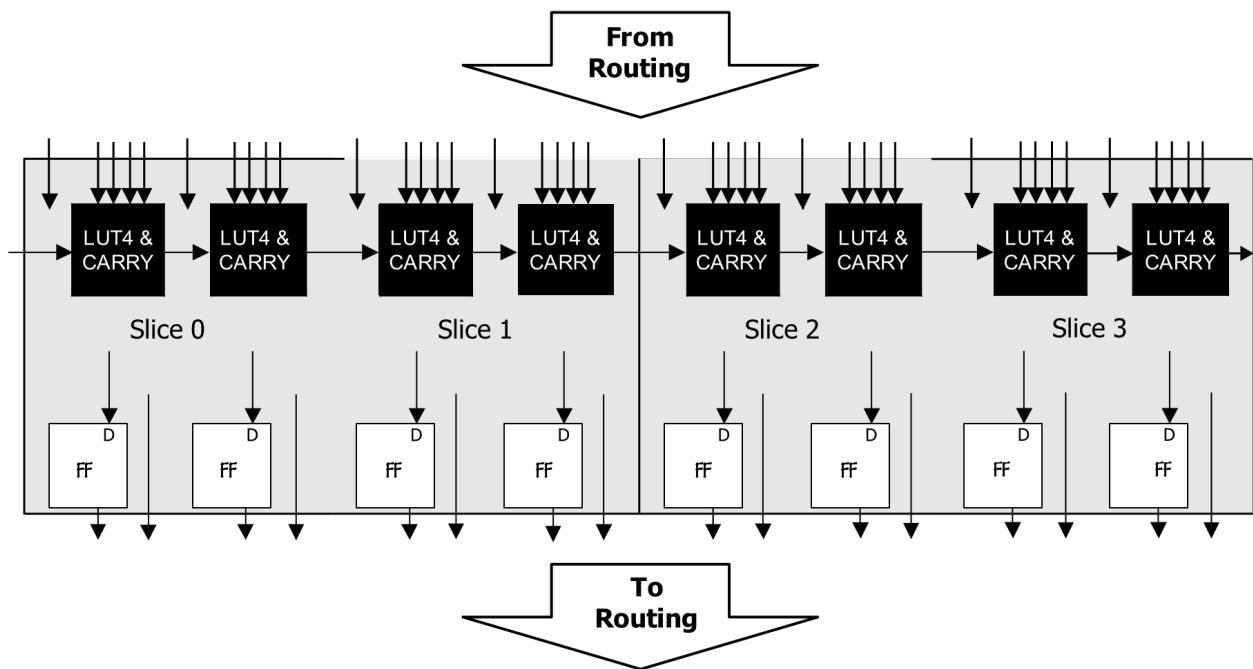


Figure 2.2. PFU Diagram

Figure 7.4: A diagram of a logic tile of a Lattice ECP5 FPGA³.

8 PERFORMANCE EXPERIMENTS

Using a machine with an Intel® Core™ i7-7700HQ CPU and 16GB DDR4 memory we conduct several experiments with different inputs and settings, the results are shown in Tables 8.2 through 8.5. For random graphs we consistently use the $G(n, M)$ Erdős–Rényi random graph model.

Path iteration	Its/success	Time/iteration	time/success	stdev	time/fail	stdev
K-Path						
DFS						
Greedy DFS						
Control point						

Table 8.1: Performance of different path iteration methods on random source graphs of size $|V| \in \{8..12\}$ with edge density 3.0 and random target graphs of size $|V| \in \{15..20\}$ with edge density 4.0. “refuse unnecessarily long paths” is enabled and TODO pruning strategy is used.

Path iteration	Its/success	Time/iteration	time/success	stdev	time/fail	stdev
K-Path						
DFS						
Greedy DFS						
Control point						

Table 8.2: Performance of different path iteration methods on random source graphs of size $|V| \in \{1..5\}$ with edge density 3.0 and a square of four ECP5 logic tiles as target graph. “refuse unnecessarily long paths” is enabled and TODO pruning strategy is used.

Source graph size	Target graph size	K-Path	DFS	Greedy DFS	CP
$ V = 5, E = 10$	$ V = 8, E = 15$	-30.0%	+10.0%	-20.0%	-17.0%
	single ECP5 tile				
	square of ECP5 tiles				

Table 8.3: The performance benefit (seconds) of the ‘refuse unnecessarily long paths’ setting on random source graphs and random target graphs.

Pruning strategy	CC	NC	Effect (succeed)	stdev	Effect (fail)	stdev
None			+0.0%	0.0	+0.0%	0.0
Empty domain						
Empty domain	×					
Empty domain		×				
Empty domain	×	×				
AllDifferent						
AllDifferent	×					
AllDifferent		×				
AllDifferent	×	×				

Table 8.4: The performance benefit (seconds) of different pruning methods with- and without Connectivity check (CC) and neighbour check (NC) on random source graphs of size $|V| \in \{8..12\}$ with edge density 3.0 and random target graphs of size $|V| \in \{15..20\}$ with edge density 4.0.

Pruning strategy	CC	NC	Par	Effect (succeed)	stdev	Effect (fail)	stdev
None			N/A	+0.0%	0.0	+0.0%	0.0
Empty domain							
Empty domain			×				
Empty domain		×					
Empty domain		×	×				
Empty domain	×						
Empty domain	×		×				
Empty domain	×	×					
Empty domain	×	×	×				
AllDifferent							
AllDifferent			×				
AllDifferent		×					
AllDifferent		×	×				
AllDifferent	×						
AllDifferent	×		×				
AllDifferent	×	×					
AllDifferent	×	×	×				

Table 8.5: The performance effect (seconds) of different pruning methods with- and without Connectivity check (CC) and neighbour check (NC) on random source graphs of size $|V| \in \{8..12\}$ with edge density 3.0 and a square of four ECP5 times as target graph.

9 SOFTWARE DESIGN

9.1 Architecture

9.2 Manual

Step 1 : Download the latest release from <https://github.com/Charrey/PathSubIsoTest>

Step 2 : Save your source graph in DOT file format, using the structure defined in Chapter TODO. If a vertex has a label, provide it with the DOT attribute key “label” and a string as value containing JSON array notation of the labels.

Step 3 : Save your target graph in the same way.

Step 4 : Open a command prompt and call:

```
java -jar <release file> <source file> <target file>
```

Optionally, use the flags:

```
-strategy [dfs/greedydfs/kpath (default)/controlpoint]
-pruning  [none (default)/zerodomain/alldifferent]
-domains  [label (default)/neighbours/reachability]
-checks   [always (default)/parallel]
-maxpaths <maximum number of paths to try between the same 2 vertices>
-maxcp    <maximum number of control points>
          (requires "-strategy controlpoint")
```

10 CONCLUSION

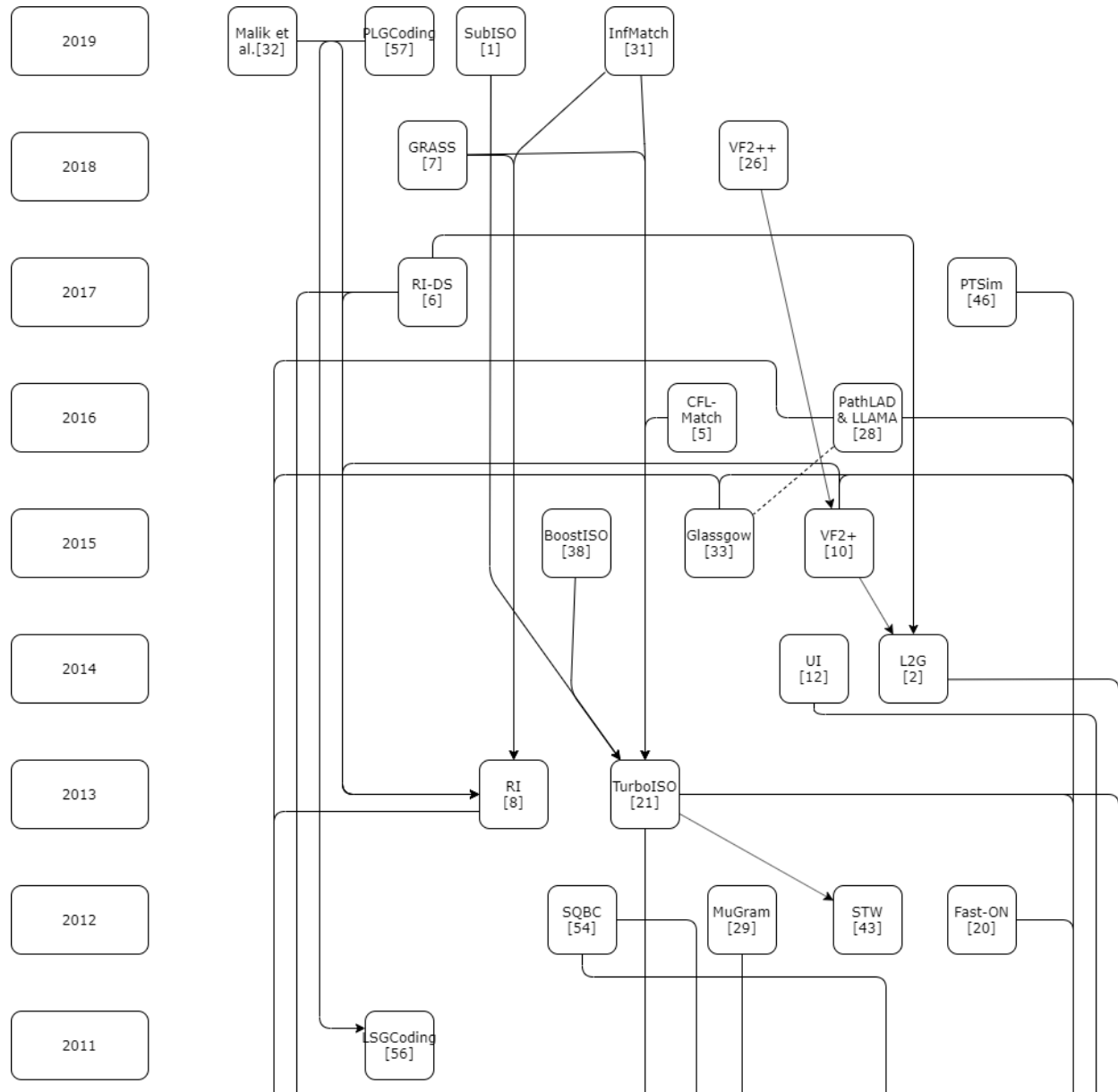
11 DISCUSSION

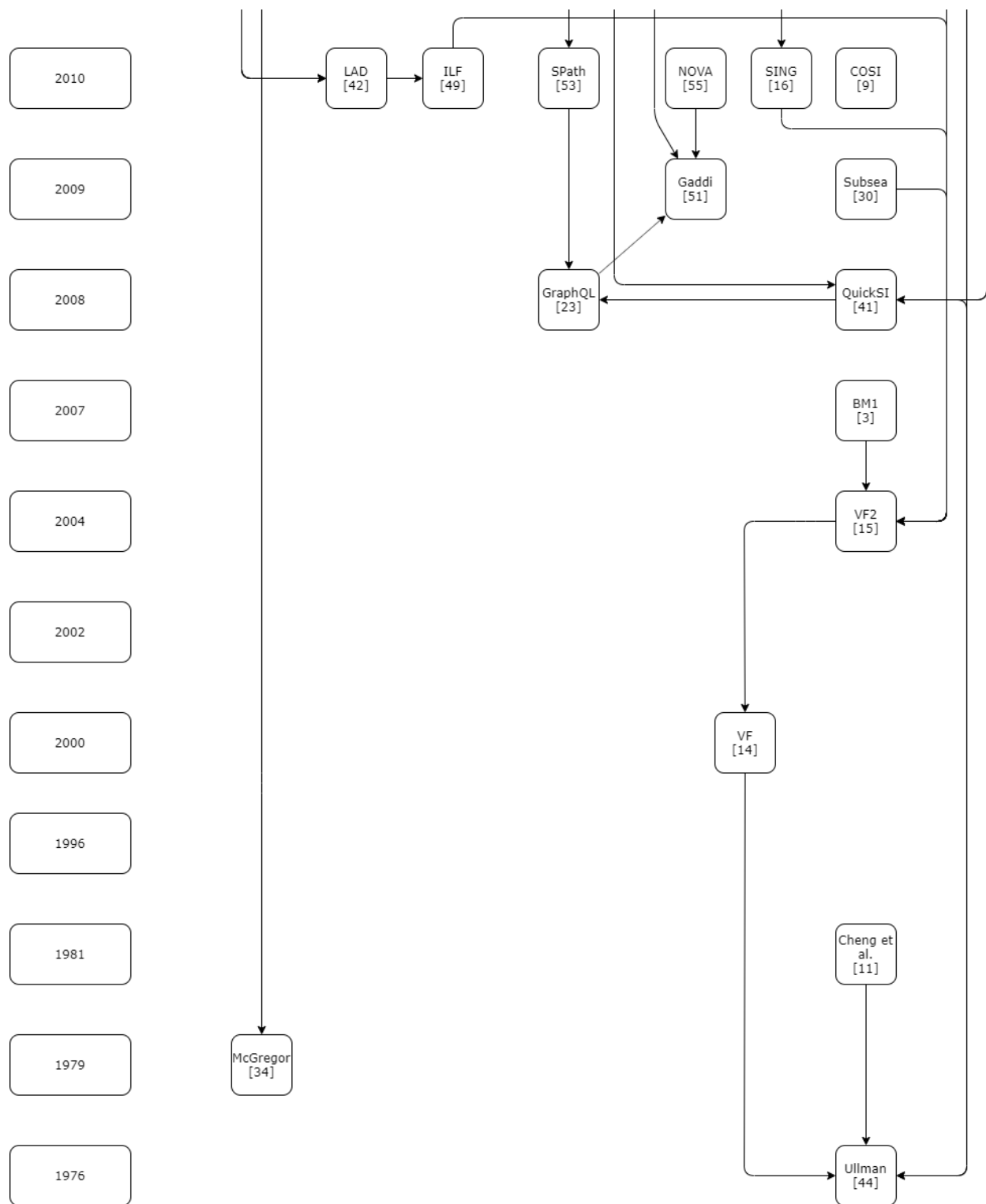
12 FUTURE RESEARCH

1. Backmarking and backjumping
2. choose target graph vertices based on some location heuristic, e.g. tiles
3. use hierarchy: e.g. when a component has been matched with some pattern Q , immediately match the next component with an isomorphic pattern Q' , allowing for backtracking

Appendices

.1 History of subgraph isomorphism algorithms





Bibliography

1. Abulaish, M., Ansari, Z.A., and Jahiruddin: SubISO: A Scalable and Novel Approach for Subgraph Isomorphism Search in Large Graph. In: 2019 11th International Conference on Communication Systems Networks (COMSNETS), pp. 102–109 (2019)
2. Almasri, I., Gao, X., and Fedoroff, N.: Quick mining of isomorphic exact large patterns from large graphs. In: IEEE International Conference on Data Mining Workshops, ICDMW, pp. 517–524 (2015)
3. Battiti, R., and Mascia, F.: An algorithm portfolio for the sub-graph isomorphism problem (2007)
4. Benz, F., Seffrin, A., and Huss, S.A.: Bil: A tool-chain for bitstream reverse-engineering. In: 22nd International Conference on Field Programmable Logic and Applications (FPL), pp. 735–738 (2012)
5. Bi, F., Chang, L., Lin, X., Qin, L., and Zhang, W.: Efficient subgraph matching by postponing Cartesian products. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 1199–1214 (2016)
6. Bonnici, V., and Giugno, R.: On the Variable Ordering in Subgraph Isomorphism Algorithms. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 14(1), 193–203 (2017)
7. Bonnici, V., Giugno, R., and Bombieri, N.: An Efficient Implementation of a Subgraph Isomorphism Algorithm for GPUs. In: Proceedings - 2018 IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2018, pp. 2674–2681 (2019)
8. Bonnici, V., Giugno, R., Pulvirenti, A., Shasha, D., and Ferro, A.: A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinformatics* 14(SUPPL7) (2013)
9. MISC
10. Brayton, R.K., Chiodo, M., Hojati, R., Kam, T., Kodandapani, K., Kurshan, R., Malik, S., Sangiovanni-Vincentelli, A.L., Sentovich, E., Shiple, T., Singh, K., and Wang, H.: BLIF-MV: An Interchange Format for Design Verification and Synthesis. Tech. rep. UCB/ERL M91/97, EECS Department, University of California, Berkeley (1991)
11. Bröcheler, M., Pugliese, A., and Subrahmanian, V.S.: COSI: Cloud Oriented Subgraph Identification in Massive Social Networks. In: 2010 International Conference on Advances in Social Networks Analysis and Mining, pp. 248–255 (2010)
12. Carletti, V., Foggia, P., and Vento, M.: VF2 plus: An improved version of VF2 for biological graphs (2015)
13. Cheng, J., and Huang, T.: A subgraph isomorphism algorithm using resolution. *Pattern Recognition* 13(5), 371–379 (1981)
14. Čibej, U., and Mihelič, J.: Search strategies for subgraph isomorphism algorithms (2014)
15. Cook, S.A.: The Complexity of Theorem-Proving Procedures. In: Proceedings of the Third Annual ACM Symposium on Theory of Computing. STOC '71, pp. 151–158. Association for Computing Machinery, Shaker Heights, Ohio, USA (1971)
16. Cordella, L.P., Foggia, P., Sansone, C., and Vento, M.: Fast graph matching for detecting CAD image components. In: Proceedings 15th International Conference on Pattern Recognition. ICPR-2000, 1034–1037 vol.2 (2000)
17. Cordella, L., Foggia, P., Sansone, C., and Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26(10), 1367–1372 (2004)

18. Di Natale, R., Ferro, A., Giugno, R., Mongiovì, M., Pulvirenti, A., and Shasha, D.: SING: Subgraph search In Non-homogeneous Graphs. *BMC Bioinformatics* 11 (2010)
19. Donath, W.E.: Complexity Theory and Design Automation. In: 17th Design Automation Conference, pp. 412–419 (1980)
20. Fortune, S., Hopcroft, J., and Wyllie, J.: The directed subgraph homeomorphism problem. *Theoretical Computer Science* 10(2), 111–121 (1980)
21. Gao, L., and Long, T.: Spaceborne Digital Signal Processing System Design Based on FPGA. In: 2008 Congress on Image and Signal Processing, pp. 577–581 (2008)
22. García, G., Jara, C., Pomares, J., Alabdo, A., Poggi, L., and Torres, F.: A survey on FPGA-based sensor systems: Towards intelligent and reconfigurable low-power sensors for computer vision, control and signal processing. *Sensors (Switzerland)* 14(4), 6247–6278 (2014)
23. Gouda, K., and Hassaan, M.: A fast algorithm for subgraph search problem. In: DE53–DE58 (2012)
24. Grohe, M., Kawarabayashi, K.-I., Marx, D., and Wollan, P.: Finding topological subgraphs is fixed-parameter tractable. In: pp. 479–488 (2011)
25. Han, W.-S., Lee, J., and Lee, J.-H.: TurboISO: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 337–348 (2013)
26. Hauck, S., and DeHon, A.: Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2007)
27. He, H., and Singh, A.K.: Graphs-at-a-Time: Query Language and Access Methods for Graph Databases. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. SIGMOD '08, pp. 405–418. Association for Computing Machinery, Vancouver, Canada (2008)
28. Al-Hyari, A.: Towards Smart FPGA Placement Using Machine Learning (2019).
29. Ikram, J., and Mohsen, M.: FPGA Implementation of a Quantum Cryptography Algorithm. *Smart Innovation, Systems and Technologies* 146, 172–181 (2020)
30. Jüttner, A., and Madarasi, P.: VF2++—An improved subgraph isomorphism algorithm. *Discrete Applied Mathematics* 242, 69–81 (2018)
31. Kanazawa, K., and Maruyama, T.: An approach for solving large SAT problems on FPGA. *ACM Transactions on Reconfigurable Technology and Systems* 4(1) (2010)
32. Karp, R.M.: On the Computational Complexity of Combinatorial Problems. *Netw.* 5(1), 45–68 (1975)
33. Kotthoff, L., McCreesh, C., and Solnon, C.: Portfolios of subgraph isomorphism algorithms (2016)
34. Krishna, V., Ranga Suri, N., and Athithan, G.: MuGRAM: An approach for multi-labelled graph matching. In: 2012 International Conference on Recent Advances in Computing and Software Systems, pp. 19–26 (2012)
35. LaPaugh, A.S., and Rivest, R.L.: The Subgraph Homeomorphism Problem. In: Proceedings of the Tenth Annual ACM Symposium on Theory of Computing. STOC '78, pp. 40–50. Association for Computing Machinery, San Diego, California, USA (1978)
36. Lingas, A., and Wahlen, M.: An exact algorithm for subgraph homeomorphism. *Journal of Discrete Algorithms* 7(4), 464–468 (2009)
37. Lipets, V., Vanetik, N., and Gudes, E.: Subsea: An efficient heuristic algorithm for subgraph isomorphism. *Data Mining and Knowledge Discovery* 19(3), 320–350 (2009)
38. Ma, T., Yu, S., Cao, J., Tian, Y., and Al-Rodhann, M.: InfMatch: Finding isomorphism subgraph on a big target graph based on the importance of vertex. *Physica A: Statistical Mechanics and its Applications* 527 (2019)
39. Malík, J., Suchý, O., and Valla, T.: Efficient Implementation of Color Coding Algorithm for Subgraph Isomorphism Problem (2019)
40. McCreesh, C., and Prosser, P.: A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs (2015)
41. McGregor, J.: Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences* 19(3), 229–250 (1979)

42. Nawari, M., Ahmed, H., Hamid, A., and Elkhidir, M.: FPGA based implementation of elliptic curve cryptography. In: Institute of Electrical and Electronics Engineers Inc. (2015)
43. Régim, J.-C.: Filtering algorithm for constraints of difference in CSPs. In: pp. 362–367. AAAI, Menlo Park, CA, United States (1994)
44. Ren, X., and Wang, J.: Exploiting Vertex Relationships in Speeding up Subgraph Isomorphism over Large Graphs. *Proc. VLDB Endow.* 8(5), 617–628 (2015)
45. Rudolf, M.: Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching. In: Ehrig, H., Engels, G., Kreowski, H.-J., and Rozenberg, G. (eds.) *Theory and Application of Graph Transformations*, pp. 238–251. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
46. Saha, S., Alam, M., and Mondol, R.: FPGA implementation of FlexRay protocol with built-in-self-test capability. In: Institute of Electrical and Electronics Engineers Inc. (2014)
47. Shang, H., Zhang, Y., Lin, X., and Yu, J.X.: Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. *Proc. VLDB Endow.* 1(1), 364–375 (2008)
48. Solnon, C.: AllDifferent-based filtering for subgraph isomorphism. *Artificial Intelligence* 174(12-13), 850–864 (2010)
49. Sun, Z., Wang, H., Wang, H., Shao, B., and Li, J.: Efficient subgraph matching on billion node graphs. In: pp. 788–799. Association for Computing Machinery (2012)
50. Ullmann, J.R.: An Algorithm for Subgraph Isomorphism. *J. ACM* 23(1), 31–42 (1976)
51. Wolf, C.: Yosys Open SYnthesis Suite. (2016)
52. Xiao, Y., Wu, W., Wang, W., and He, Z.: Efficient Algorithms for Node Disjoint Subgraph Homeomorphism Determination. In: Haritsa, J.R., Kotagiri, R., and Pudi, V. (eds.) *Database Systems for Advanced Applications*, pp. 452–460. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
53. Xie, X., Li, Z., and Zhang, H.: Efficient Subgraph Matching in Large Graph with Partitioning Scheme. In: *Proceedings - 13th Web Information Systems and Applications Conference, WISA 2016 - In conjunction with 1st Symposium on Big Data Processing and Analysis, BDPA 2016 and 1st Workshop on Information System Security, ISS 2016*, pp. 28–33 (2017)
54. Yalla, P., and Kaps, J.-P.: Lightweight cryptography for FPGAs. In: pp. 225–230 (2009)
55. Yu, H., Lee, H., Lee, S., Kim, Y., and Lee, H.-M.: Recent advances in FPGA reverse engineering. *Electronics (Switzerland)* 7(10) (2018)
56. Zampelli, S., Deville, Y., and Solnon, C.: Solving subgraph isomorphism problems with constraint programming. *Constraints* 15(3), 327–353 (2010)
57. Zhang, S., Hu, M., and Yang, J.: TreePi: A novel graph indexing method. In: *Proceedings - International Conference on Data Engineering*, pp. 966–975 (2007)
58. Zhang, S., Li, S., and Yang, J.: GADDI: Distance Index Based Subgraph Matching in Biological Networks. In: *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology. EDBT '09*, pp. 192–203. Association for Computing Machinery, Saint Petersburg, Russia (2009)
59. Zhang, T., Wang, J., Guo, S., and Chen, Z.: A Comprehensive FPGA Reverse Engineering Tool-Chain: From Bitstream to RTL Code. *IEEE Access* 7, 38379–38389 (2019)
60. Zhao, P., and Han, J.: On Graph Query Optimization in Large Networks. *Proc. VLDB Endow.* 3(1–2), 340–351 (2010)
61. Zheng, W., Zou, L., Lian, X., Zhang, H., Wang, W., and Zhao, D.: SQBC: An efficient subgraph matching method over large and dense graphs. *Information Sciences* 261, 116–131 (2014)
62. Zhu, K., Zhang, Y., Lin, X., Zhu, G., and Wang, W.: NOVA: A novel and efficient framework for finding subgraph isomorphism mappings in large graphs. *Lecture Notes in Computer Science (including sub-series Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5981 LNCS(PART 1), 140–154 (2010)
63. Zhu, L., and Song, Q.: A study of Laplacian spectra of graph for subgraph queries. In: pp. 1272–1277 (2011)
64. Zhu, L., Yao, Y., Wang, Y., Hei, X., Zhao, Q., Ji, W., and Yao, Q.: A novel subgraph querying method based on paths and spectra. *Neural Computing and Applications* 31(9), 5671–5678 (2019)