University of Twente
EEMCS
Formal Methods and Tools

FPGA-on-FPGA emulation using 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 this research, we establish a methodology for finding such emulation mappings. To this end, we adapt Xiao's algorithm for subgraph homeomorphism and optimize it for usage with graphs representing FPGAs. This algorithm aims to find an emulation in as many cases as possible, as quickly as possible. Based on experiments run using this algorithm, we evaluate different settings for our algorithm and establish an optimal configuration set for FPGA emulation graphs. Using this configuration set we show that subgraph homeomorphism is computationally and space-wise feasible for FPGA emulation problems. The result is a software package that computes FPGA emulators: programs whose output is a program for a concrete FPGA that emulates the provided input program for the virtual FPGA.

CONTENTS

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 [21], cryptography[56, 43, 29], digital signal processing[20], protocol implementations[48] 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, which is the industry standard used for testing ASICs (non-programmable FPGAs). 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 and thus a negative learning experience.

Another technique that could be used to execute an FPGA configuration is modelling the semantics of virtual FPGA components in an HDL module. The engineer can then simulate the execution of a virtual 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 3.3 and Figure 3.7. 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. This technique is generally not used in the industry because of this reduced performance.

Techniques that require students to implement FPGA programs in HDLs furthermore abstract away the process of place & route, a process that significantly affects the perfor-

mance of the hardware implementation of the program. Since students should be educated to use this process optimally or even improve it, they should be aware of it 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. This way, the course designer has full control over the learning environment without being constrained to physical hardware. Performing the NP-complete place & route manually allows the rest of the compilation process to take linear time, providing a configuration for the virtual FPGA. We aim to devise a method to compile this configuration for a virtual FPGA to a configuration compatible with a concrete one in linear time.

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). This mapping retains the semantics of the configuration but makes it suitable for execution on real hardware. While the format of an FPGA configuration is usually kept secret by hardware vendors [25, 62, 4], some FPGA boards have been reverse engineered to reveal the underlying format [58, 44]. 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.

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.

The goals of this research are specified in Chapter 2 (Objectives). This is a more specific description of the problem we outlined in this introduction requiring little background information. In Chapter 3 (Background) we will provide background on FPGAs and the graph problem 'subgraph homeomorphism': a graph problem we will use to reach our objectives. In Chapter 5 (Models) we specify how we model FPGAs as graphs to make them applicable for subgraph homeomorphism. In Chapter 6 (Algorithm) we describe the algorithm we use to solve the subgraph homeomorphism with these graphs. We prune the

search space with methods described in Chapter 7 (Pruning). We perform experiments with models and the algorithm with different settings in Chapter 10 (Experiments), followed by a discussion (Chapter 11) interpreting the results and a conclusion in Chapter 12.

2 OBJECTIVES

This research involves 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 and a graph specification of the structural layout of a concrete FPGA B, how do you assemble a linear[1] 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 models of both a virtual FPGA and a (larger) concrete FPGA, generates a function that translates a model of an FPGA program compatible with the virtual FPGA to a model of an FPGA program compatible with the concrete FPGA. We generalise this algorithm to other use cases with the same underlying mathematical problem (node disjoint subgraph homeomorphism). Furthermore, we include specifications on how to model FPGA models for this purpose.

[1] i.e. a numeric constant c exist such that the number of instructions required in the execution of f is less than $c * (\text{the number of configurable components of the virtual FPGA})$. Note that this depends on the size of the *unconfigured* virtual FPGA, not on the size of the user-provided configuration of said FPGA.

3 BACKGROUND

3.1 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 sequentially loading instructions. This computation process implies that FPGAs are very efficient at executing concurrent programs. Recon-

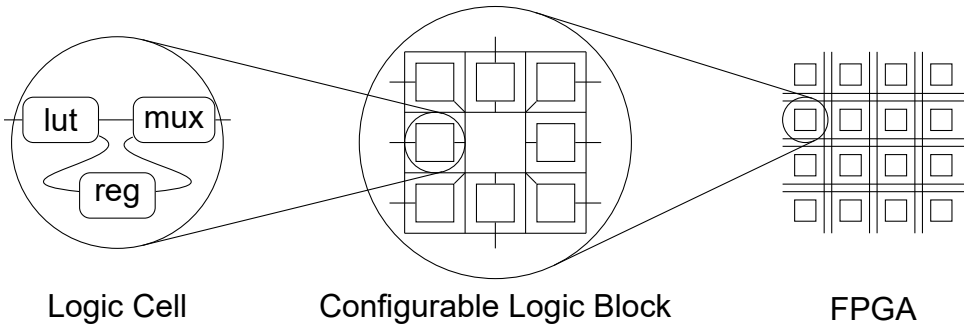


Figure 3.1: The hierarchy of a typical FPGA. A typical FPGA mostly consists of Configurable Logic Blocks (CLBs) and a CLB mostly consists of logic cells. The way logic cells and CLBs are connected may be different for each FPGA architecture.

p	q	p XOR q
F	F	F
F	T	T
T	F	T
T	T	F

Figure 3.2: A truth table that shows the result of an XOR operation

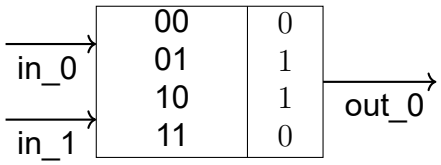


Figure 3.3: A Lookup Table (LUT) in which an XOR-operation is configured

figuring 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. Moreover, FPGAs can be slower than CPUs for nonparallel applications since its longer critical path length (i.e. distance an electric current has to travel each clock cycle) results in a lower clock speed. Lastly, the different structure of an FPGA requires programs for them to be developed in specialized HDL languages instead of CPU-based programming languages, although efforts are being made to bring these domains closer together.

FPGAs perform execution using components such as lookup tables, registers, and special-purpose modules[1]. These components are physical structures on the circuit. In the next sections, we will discuss these modules.

3.1.1 Lookup tables

Any boolean logic formula can be expressed in the form of a truth table, such as in Figure 3.2. 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 3.3 shows a lookup table that has two

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

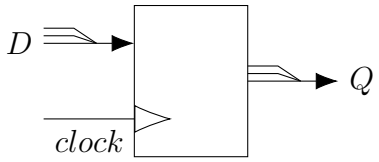


Figure 3.4: Traditional representation of a register. Note that input *D* and output *Q* may consist of multiple wires.

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.

3.1.2 Registers

Programs require intermediate data storage to perform any computation that is more complex than combinational logic.

FPGAs use registers for this purpose (see Figure 3.4). 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 clocks- wires whose signal constantly changes between 0 and 1 that is connected to registers in the hardware. This can be a single global clock connected to each register or a network of clocks each responsible for different parts of the FPGA. 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.

3.1.3 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

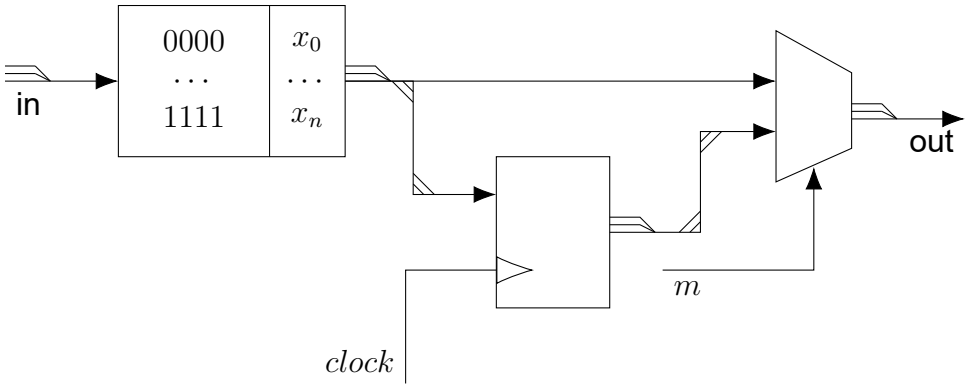


Figure 3.5: An n-input logic cell. x_k and m must be configured for any $0 \leq k \leq n$

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 3.5. Unfortunately for us, vendors have different logic cell designs, meaning we cannot generalize this example.

FPGA manufacturers often group Logic cells in Configurable Logic Blocks (CLB) that make hardware production, placement, and routing (Section 3.2) 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.

3.1.4 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 transistors. These are tiny modules on an FPGA board that are connected with an input wire and an output wire. They can be configured to allow electric current flowing from its input to its output or to block any incoming electric current.

3.1.5 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.

3.2 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 an NP-hard problem while finding any matching routing configuration is an NP problem. Both can be reducted to the problem of fixed-vertex subgraph homeomorphism of which the optimisation variant needs to explore the entire search tree, which [37] showed can be computed in $O(2^{|T|-|S|}n^{O(1)})$ for the general case. However, routing algorithms are often made for specific FPGA architectures, allowing for better heuristics and thus performance.

! Note that place & route of an FPGA *program* as described here is a very similar problem to the emulation an unconfigured virtual FPGA to a concrete FPGA (this research). However, it is important to note that these are different problems: with place & route, a mapping is obtained from a model of an *FPGA program* to the concrete FPGA. In our research, we obtain a mapping from a model of a *virtual FPGA* to the concrete FPGA. The two problems are on different levels of abstraction: an FPGA program does not have the concept of unconfigured components or transistors 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 that 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. They are usually modular with components that have relatively few interconnecting wires. FPGAs, however, can consist of densely connected components that can severely impact routeability. We leave the analysis of place & route algorithms for compatibility with FPGA-on-FPGA emulation for future research.

3.3 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. This way, the *static* configuration of the virtual FPGA uses the *dynamic* resources (e.g. RAM and registers) of the concrete FPGA. A simulation of a single LUT is shown in Figure 3.7.

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 dynamic components on the FPGA need to be used to store and execute the configuration of the same program.

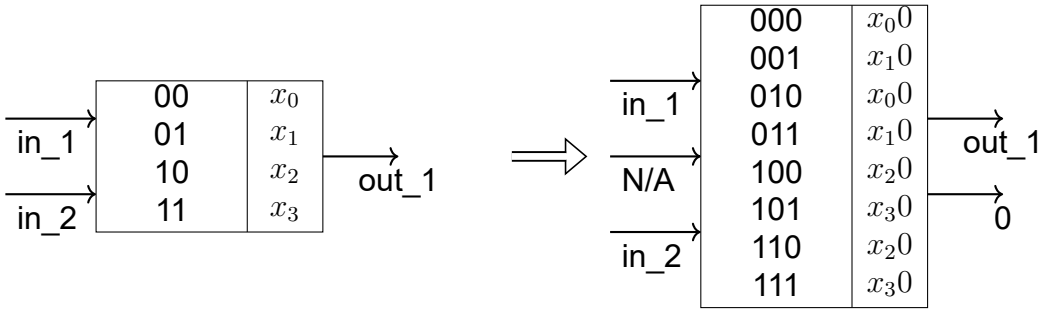


Figure 3.6: Emulation of an FPGA: the configuration of the virtual FPGA is reflected in the configuration of the concrete FPGA. Any extra inputs and outputs are unused.

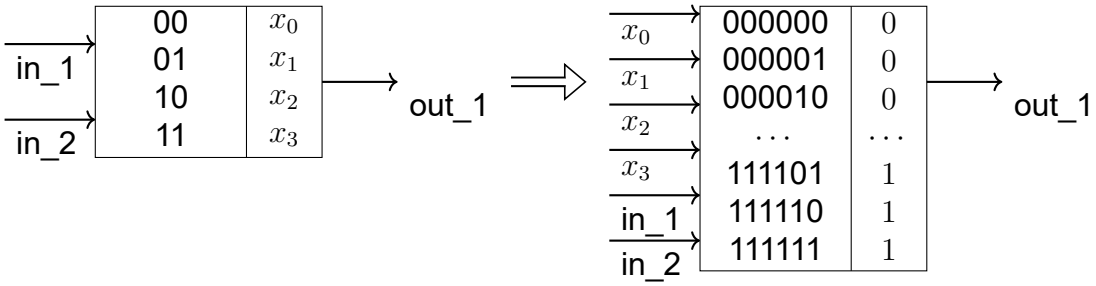


Figure 3.7: Simulation of an FPGA: the programming of the virtual FPGA is reflected in *additional input* of the actual FPGA originating from some sort of explicit configuration source, e.g. RAM or a storage device connected to the concrete FPGA's input pins. The configuration of the concrete FPGA does *not* depend on the configuration of the virtual FPGA and additional logic- and storage resources are used.

3.4 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 3.6 shows the emulation counterpart of Figure 3.7 with a smaller LUT. Since the target FPGA has one more input than the virtual FPGA, that input remains unused. Similary, since it has one more output, that output remains unused (i.e. always outputs 0). The configuration of the LUT is such that it has the same semantics as the virtual LUT with these constraints in mind. 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.

3.5 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.

A graph representation of the physical structure of FPGAs allows us to scan through the structure of the concret FPGA graph and look for structures that resemble the graph of the virtual FPGA. Let us, for example, assume that the structure of the concrete FPGA graphs contains a complete embedding of the virtual FPGA graph. Then, by disabling each component outside of this embedding and by copying configurations from components in the virtual FPGA model to their respective components in the concrete FPGA graph model we would have an emulation function. This could entail changing some bits if, for example, the second output of a virtual LUT is mapped the the first output of the concrete LUT and vica versa.[2]

The graph theoretic name for finding these embeddings is subgraph isomorphism. It is an NP-complete problem[15] 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 an 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 (vertex disjoint) subgraph homeomorphism. In this problem, intermediate vertices are allowed in the embedding on the target-graph side.

[2]While virtual components may also be emulated by structures of multiple concrete components connected in specific ways, we deem finding such emulations out of scope for this research.

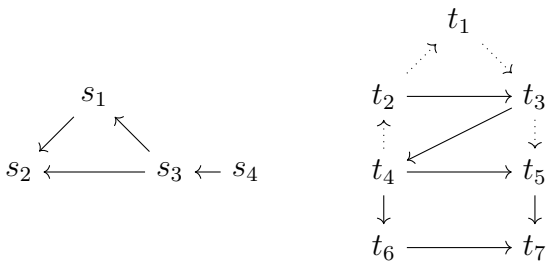


Figure 3.8: Two graphs G_1 and G_2. G_1 is (node disjoint) subgraph homeomorphic to G_2 with the mapping $\{(s_1, t_5), (s_2, t_7), (s_3, t_4), (s_4, t_2), (s_1 \to s_2, t_5 \to t_7), (s_3 \to s_1, t_4 \to t_5), (s_3 \to s_2, t_4 \to t_6 \to t_7), (s_4 \to s_3, t_2 \to t_3 \to t_4)\}$. Other homeomorphisms exist as well.

This means that the embedding consists of a vertex-to-vertex mapping and an edge-to-path mapping.

This is the essence of our research: finding some subset of the concrete FPGA (using graphs) that is topologically the same way as the virtual FPGA and has the appropriate components that can emulate virtual components one-to-one.

> **Definition 3.5.1** (graph)**.** A graph is a tuple (V, E, L) where V is a set of vertices, E is a multiset of directed edges such that each edge $e \in E \to e \in (V \times V)$ and $L : (V \to \mathbb{P}(\lambda))$ is a multilabelling function where λ is a finite alphabet.
>
> **Definition 3.5.2** ((vertex disjoint) subgraph homeomorphism)**.** Let P the set of all loopless paths in G_T, 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 subgraph homeomorphism from graph S to graph T is a tuple $(\textit{vmap}, \textit{emap})$ where ***vmap*** $\subseteq (V_S \mapsto V_T)$ and ***emap*** $\subseteq (E_S \mapsto P)$ are both injective functions such that:
>
> 1. $\forall s \in V_S . L_S(s) \subseteq L_T(\textit{vmap}(s))$
>   i.e. vertices are mapped to vertices that have at least the same label set.
> 2. $\forall (s_1, s_2) \in E_S . \textit{first}(\textit{emap}(s_1, s_2)) = \textit{vmap}(s_1) \wedge \textit{last}(\textit{emap}(s_1, s_2)) = \textit{vmap}(s_2)$
>   i.e. each edge in G_1 is mapped to a path in G_2.
> 3. $\forall p \in \textit{values}(\textit{emap}) . \forall x \in \textit{intermediate}(p) .$
> $\nexists p' \in (\textit{values}(\textit{emap}) \setminus \{p\}) . x \in \textit{intermediate}(p')$
>   i.e. these paths are internally vertex disjoint.
>
> This tuple is also the certificate for the decision problem of subgraph homeomorphism, which returns whether a tuple like this exists. A popular different way to define the decision problem is whether the target graph contains a subgraph which can be obtained by repeatedly intersecting the source graph's edges with vertices.

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 emulation[3][4]. The vertex-to-vertex mapping *vmap* describes how to obtain configurations for the concrete FPGA and the edge-to-path mapping *emap* describes how vertices in the concrete FPGA are connected via paths of intermediate components configured 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.

[3] If no such relation exists, it does not mean a function f as specified in Section 2 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.

[4] This 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.

4 LITERATURE

4.1 FPGA compilation

Yosys[53] 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.
- 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.

4.2 Subgraph isomorphism

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”. We examined matching papers and selected those that solve exact (not approximate) subgraph isomorphism 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.

Malik et al.[40] simplify subgraph isomorphism by repeatedly randomly simplifying the target graph using a colouring.

PLGCoding[67], based on LSGCoding[66] 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[52] to search these subregions for the subgraph. Similarly, InfMatch[39] uses heuristics to select a node in the target graph and selects subregions in the target graph from that node to search in. PTSim[55] 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[47] 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[50] solves subgraph isomorphism using constraints. It then applies AllDifferent filtering (using an algorithm by Régin[45]) 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[42] 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[46] 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[41] 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[24] 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[64] 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[51] 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[26] 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 [59] 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[65] 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[60] and Gaddi[61] 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[63] also uses indexing techniques on trees and subgraphs to speed up the search for a complete mapping.

Subsea[38] recursively cuts the target graph along its minimal cut, searches for the subgraph on the edges on this cut and the 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[49] 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[52] representation of partial mappings.

Ullmann[52] 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[22] 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[52] 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 subgraph homeomorphism.

4.3 Subgraph homeomorphism

We performed literature research using the same method as for subgraph isomorphism but with the terms "subgraph homeomorphism" and "topological minors". We found the following existing research:

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

Grohe et al.[23] 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.

[1]The algorithm involves precomputing all possible paths. The number of possible paths increases exponentially with the sizes of both the pattern- and target graph. Applying this to a graph model of an FPGA is infeasible.

5 MODELS

Our algorithm will generate an emulation using vertex disjoint subgraph homeomorphism. To do this, we model both the virtual and concrete FPGA as vertex-multilabeled directed graphs as defined in Definition 3.5.1. This section specifies this process.

In short, we model each logic cell and each wire and transistor that are not part of a logic cell as vertices, and add an extra vertex for each input- and output of logic cells. The edges between the vertices denote either the direction of a transistor (if connecting a transistor and a wire) or the data flow of a logic cell (if connecting a wire with an in/output or an in/output with a logic cell).

We label each wire vertex with the label `WIRE`, and the additional label `EDGE` if they function as input- or output of the entire FPGA. We label each transistor with the label `ARC`, each logic cell with the label `SLICE` and each in/output with the label `PORT`. Furthermore, we add the label `CE` to an input that enables writing data to the register of the logic cell. Lastly, we add the labels $\{$`CONFIGURABLE`, `UNCONFIGURABLE`$\}$ to each transistor that is configurable by the user, and the label `UNCONFIGURABLE` to transistors that are always enabled and not configurable by the FPGA configurator. We use these labels to to avoid unintended electrical current flow between parts in the concrete FPGA. More information on how we use this label within the context of our algorithm can be found in Section 6.7.

6 ALGORITHM

6.1 Baseline: ndSHD2

In our research we will extend existing work on node disjoint subgraph isomorphism. In the literature we found two existing well-defined algorithms: Xiao's algorithm 'ndSHD2' [54] and Lingas algorithm [36]. We choose to adapt Xiao's algorithm ndSHD2 as baseline over Lingas because it is simpler, includes a method for calculating a vertex-on-vertex mapping and has many similarities with subgraph isomorphism algorithms. In comparison, Lingas' algorithm suggests attempting their edge-path mapping strategy on every possible vertex-on-vertex mapping. We will adapt Xiao's algorithm to compute paths on-the-fly instead of beforehand to avoid the exponential space requirement.

To understand ndSHD2 better, let us first examine what the output is supposed to be. The algorithm should, given two graphs, not just indicate whether the latter graph has a subgraph homeomorphism of the former, but also indicate what subgraph homeomorphism that is. More formally, we need to solve the *certification* problem rather than the *decision* problem. This certification indicates to which target graph vertex each source graph vertex is mapped and to which target graph path each source graph edge is mapped such that the entire mapped source graph is a subgraph of the target graph. This is what we will from now on refer to as the **mapping**. If, during the execution, the mapping is not yet complete, i.e. it does not have a target for some source graph vertex or edge, we refer to it as the **partial mapping**.

ndSHD2 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.

In our literature study, we found that the core of many subgraph isomorphism algorithms[1] is a form of similar DFS state space exploration where the states consist of partial vertex-to-vertex mappings. In subgraph isomorphisms, the edge-to-edge mapping is elementary by performing the vertex mapping on the edge source and target to obtain the target edge. With the subgraph homeomorphism algorithm such as Xiao's, an additional mapping from

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

Algorithm 1: ndSHD2

Inputs: the current vertex-vertex partial mapping *vmap*, the current edge-path *emap* partial mapping
Outputs: *found*, indicating whether a valid subgraph homeomorphism has been found.
if s *is complete* **then**
    return true;
end
found $\longleftarrow$ *false*
while !*found* $\wedge \exists$ *valid node/edge-path mapping pair* **do**
    if *hasUnmatchedEdge*() **then**
        $(e_S, p_T) \longleftarrow$ *GetNextEdgePathPair*();
        if *wouldPrune*(*vmap*, *emap* $\cup \{(e_S, p_T)\}$) **then**
            continue;
        end
        emap $\longleftarrow$ *emap* $\cup \{(e_S, p_T)\}$;
        found $\longleftarrow$ *ndSHD3*(*vmap*, *emap*);
        if *found* **then**
            return true;
        end
        emap $\longleftarrow$ *emap* $\cap \{(e_S, p_T)\}$;
    end
    else
        $(v_S, v_T) \longleftarrow$ *GetNextNodePair*();
        if *wouldPrune*(*vmap* $\cup \{(v_S, v_T)\}$, *emap*) **then**
            continue;
        end
        vmap $\longleftarrow$ *vmap* $\cup \{(v_S, v_T)\}$;
        found $\longleftarrow$ *ndSHD3*(*vmap*, *emap*);
        if *found* **then**
            return true;
        end
        vmap $\longleftarrow$ *vmap* $\cap \{(v_S, v_T)\}$;
    end
end
return false;

Algorithm 2: wouldPrune

Inputs: the current vertex-vertex partial mapping *vmap*, the current edge-path *emap* partial mapping

Outputs: whether pruning should be applied.

domains $\longleftarrow$ *filterDomains*(*vmap*, *emap*);

return $\emptyset \in$ *values*(*domains*);

E_S to the path set of G_T is needed.

The algorithm starts with an empty partial mapping, and starts mapping source graph vertices to target graph vertices. Whenever both vertices of a source graph edge have been matched, the algorithm finds an arbitrary path between the mapped edge source and target in the target graph that is internally disjoint from target graph vertices already used in the partial mapping. The algorithm prioritizes extending the matching with an edge-path pair over extending it with a vertex-vertex pmair. Whenever no such path or vertex exists, the algorithm undoes (backtracks) the last matching step taken and tries a different alternative. Whenever a partial match is extended with a vertex-vertex pair or with an edge-path pair the search space is refined using a pruning strategy implemented by the $wouldPrune$ call. This method filters out future path candidates from a path storage and from a vertex storage according to zero-domain N-reachability pruning (see Chapter 7 for definitions). This removes path candidates and vertex candidates that can provably not be part of any valid subgraph homeomorphism mapping.

The algorithm as specified in Xiao's paper leaves out some specific ordering details. Because these orderings are unknown, we specify our own orderings in Sections 6.3, 6.4. We will explain what ordering of paths we use in Section 6.5. We will introduce some optimisations to the algorithm in Section 6.6.

6.2 How to choose vertex-vertex pairs

There are different ways to choose which vertex-vertex pair to try adding to the partial mapping first, and which only to try adding after other attempts have failed. These methods can be divided into three categories:

1. Choose a source graph vertex first using some heuristic, then attempt all target graph vertex candidates using another heuristic.
2. Choose a target graph vertex first using some heuristic, then attempt all source graph vertex candidates using another heuristic.
3. Choose a source vertex-target vertex with high heuristic first, then attempt all other pairs.

While the strategy used does not affect the vertex-on-vertex mapping state space, it can

affect the average-case performance. Strategy 3, for example, uses combined heurstics that may be more powerful. The disadvantage of it requires $O(|V_s| * |V_t|)$ heuristic computations before the first pair is obtained. If these computations are done beforehand the heuristic cannot take the current partial mapping into account and is likely to be weak. Is this computation is done at each step in the search process, it introduces considerable delay.

Strategies 1 and 2, on the other hand, require only $O(|V_s| + |V_t|)$ heuristic calculations to be made before obtaining a candidate pair. We choose strategy 1 with a source graph vertex heuristic that is precomputed and provide two heuristics for target graph vertices, one of which is precomputed and one of which is calculated run-time.

6.3 Source graph vertex order

The source graph vertex order is the order in which source graph vertices are added to the partial mapping. For example, if $s_1 \prec_M s_2$ in this ordering, then a pair with s_2 and some target graph vertex will only be added to the partial mapping if a pair with s_1 is already in it within the context of a partial matching M. If this ordering depends on M, it has to be calculated at run-time. If it does not, it only has to be precomputed once.

Xiao does not specify a source graph vertex order. Instead, we will use a high-performance ordering technique from the subgraph *isomorphism* domain. From performance comparisons (see Appendix .1) we extract that the best performing algorithm that adheres to partial mapping search for subgraph isomorphism is RI-DS [8]. In our literature study, we found no evidence a faster algorithm exists. This ordering does not depend on M, and thus we will precompute the entire ordering.

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

> **Definition 6.3.1** (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 6.3.2** (neighbour set)**.** If some $v \in V$, then ***neighbours***$(v) :=$ ***succ***$(v) \cup$ ***pred***(v).

The algorithm RI-DS obtains a static source graph vertex ordering using a greedy algorithm called “GreatestConstrainedFirst”. This algorithm starts with an empty list and adds vertices to the end of the list, resulting in an ordering in which vertices earlier in the list are prioritized. It starts with a list μ containing only the source graph vertex with the highest degree, i.e. $s \in V_S$ where $|$***neighbours***$(s)|$ is highest. Then, each source graph vertex not yet in the list is assigned three scores N_1, N_2 and N_3. $N_1(s)$ is the number of neighbours

of s that are already in the list (i.e. $|\{s'|s' \in \textit{neighbours}(s) \wedge s' \in \mu\}|$). $N_2(s)$ is the number of neighbours of s that are not in the list themselves, but do have a neighbour in the list (i.e. $|\{s'|s' \in \textit{neighbours}(s) \wedge s' \notin \mu \wedge \textit{neighbours}(s') \cap \mu \neq \emptyset\}|$). $N_3(s_x)$ is the number of all remaining neighbours of s (i.e. $|\{s'|s' \in \textit{neighbours}(s) \wedge s' \notin \mu \wedge \textit{neighbours}(s_y) \cap \mu = \emptyset\}|$). It selects each vertex s of which $N_1(s)$ is greatest. Ties are broken with the greatest N_2-value, and any remaining ties are broken with the greatest N_3-value. Any remaining ties are broken randomly. The selected vertex is added to the back of the list, and the process repeats. This continues until every vertex has been added to the list.

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

6.4 Target graph vertex order

The target graph vertex order is the order in which target graph vertices are added to the partial mapping. For example, if $t_1 \prec t_2$ in this ordering, then a pair with t_2 and some source graph vertex s_x will only be added to the partial mapping if the pair (s_x, t_1) was already proved unfeasible. Xiao does not describe a specific target graph vertex order.

We found one subgraph isomorphism algorithm that describes a specific target graph vertex order, being Glasgow [41]. In this algorithm, target graph vertices with higher degree are prioritized. Since this does not depend on the chosen source graph vertex or on the current partial matching, this order can be precomputed, introducing only linear complexity. Formally, when some source graph vertex s needs to be matched, target graph vertices are chosen using the following metric, choosing vertices with lower metric values first:

$$\textit{metric}_{\textit{degree}}(s,t) = -|\textit{neighbours}(t)|$$

Another option (that is not attempted by subgraph homeomorphism algorithms before) is to take the current partial mapping into account. Using a degree-based or random target graph vertex order will result in chosen target graph vertices to be independent of already used target graph vertices, potentially requiring many resouces to reach. However, we can use information from the current partial mapping to obtain better candidates: since we know the chosen source graph vertex and thus which edges will be mapped after this vertex-vertex pair, we can choose our target vertex such that the paths associated with these edges are as short as possible.

Whenever we need to match some source graph vertex s, This distance-based method will choose target vertices first that have the lowest distance to the source graph vertex' neighbours that are already in the partial matching. The result is that fewer number of vertices

need to be used as intermediate vertex in the paths associated with the edges to those neighbours. The disadvantage of this method is that it requires runtime usage of a computionally expensive shortest path algorithm to choose a target graph vertex candidate. The shortest path algorithm can be cached, which reduces the number of computations needed after backtracking but introduces quadratic ($O(|V_s| * |V_t|)$) space usage (we implement this with- and without caching). Formally, when some source graph vertex s needs to be matched, t_x are chosen using the following metric, choosing vertices with lower metric values first:

$$\mathit{metric}_{\mathit{distance}}(s,t) = \sum_{s' \in E(s)} \begin{cases} |\mathit{shortestPathUndirected}(M(s'),t)| & s' \prec_\mu s \\ 0 & s \prec_\mu s' \end{cases}$$

Here, M is the current partial mapping and μ is the source graph vertex order. We implemented both methods to use in our algorithm.

6.5 Path iteration

In subgraph isomorphism, edge-on-edge mappings can be trivially computed 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 loopless paths from shortest to longest, avoiding unusable vertices in the existing partial mapping. We use Yen's algorithm [57] for this. Faster and more recent algorithms exist [27, 9] but lack public implementations.
- **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 (graph distance) -** Search for paths using greedy depth first search, choosing the direction closest to the goal vertex first (avoiding unusable vertices in the existing partial mapping). We implement both a variant that precomputes all shortest paths and a variant that calculates at run-time which direction to choose (without caching).
- **Greedy DFS (informed graph distance) -** Search for paths using greedy depth first search, choosing the direction closest to the goal vertex along a path that avoids unusuble vertices first (avoiding unusable vertices in the existing partial mapping). We compute run-time which direction to choose.
- **Control point -** Select increasingly many 'control points' (from 0 to $|V|$) randomly in the target graph that must be in the path in a specific order, then connecting them by a

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

Table 6.1: Worst-case space complexity of each path iteration strategy. The computational requirements of each method change in different ways for subsequent calls.

shortest path algorithm that avoids unusable vertices in the existing partial mapping. We implemented this with a recursive algorithm in which a replacement of some control point is only attempted if all control points *earlier* in the control point order have been attempted.[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 6.1, and examples of paths returned by them are shown in Appendix 2.

6.6 Optimisations

Since our use case entails finding subgraph homeomorphisms in relatively large graphs, any optimisation possible improves our chances of finding FPGA emulation mappings within reasonable time. Therefore, in addition to implementing ordering parameters for the ndSHD2 algorithm, we implement some optimisations and individually evaluate them. The algorithm with optimisations implemented is shown in Algorithm 3.

6.6.1 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 $t_0 \dots t_n$ is "unnecessarily long" iff:

$\exists t_i \in t_0 \dots t_{n-2}. \exists t_j \in t_{i+2} \dots t_n. (t_i, t_j) \in E_T$

With this optimisation, such paths are skipped by path iterators. Examples of this effect are shown in Appendix .2, Figures 2 and 3.

[2] To avoid duplicate paths, any path that can be generated with fewer control points is skipped. Furthermore, any control point configuration where shifting some control point towards the goal vertex along the path results in the same path is skipped.

6.6.2 Runtime Pruning

Some subgraph isomorphism algorithms [17, 41] prune the search space during the search using some detection method of dead search paths. Xiao's algorithm does this as well. For our algorithm, we will implement Xiao's pruning methods and weaker (but more less computationally demanding) variants. In Chapter 7 we will elaborate on these methods.

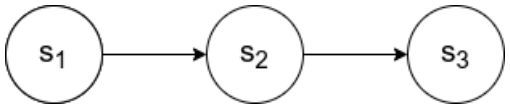
6.6.3 Contraction

The source graph of a homeomorphism case will often contain vertices that have exactly one incoming edge and one outgoing edge. In FPGAs, for example, these could be transistors or ports of components. These vertices do topologically nothing but serve as an edge; therefore they may also be thought of as an edge, where the edge source is the vertex' predecessor and the edge target is the vertex' successor. When we use the word contraction, we mean surpressing all such vertices in the virtual FPGA and replacing them with a single edge, as illustrated in Figure 6.1. This process transforms the source graph S into the smallest graph S_{cont} that is topologically equivalent to S. This may be a multigraph or a graph that contains self-loops. If a subgraph homeomorphism exists from S to some target graph T, then there also exists a subgraph homeomorphism from S_{cont} to T (a proof is given in Appendix .3). The optimisation 'contraction' searches for a homeomorphism between S_{cont} and the T instead of finding one between S and T. Because this involves a smaller source graph, it also involves a smaller search space. Note that we do not change/contract the target graph representing the concrete FPGA.

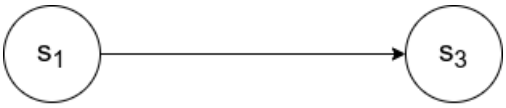
The process of finding a homeomorphism from S_{cont} to T is slightly different from what normal. This is because while every homeomorphism from S to T can be deduced to a homeomorphism between S_{cont} and T (by contraction of vertices and concatenation of paths), not every homeomorphism from S_{cont} and T allow a homeomorphism from S to T to be inferred. An example is the case where S_{cont} is isomorphic to T and S subdivides a single edge of S_{cont}: there exists a subgraph homeomorphism from S_{cont} to T since they are isomorphic, but there is no subgraph homeomorphism from S to T. Because of this, we need to adapt the subgraph homeomorphism search process such that it only finds subgraph homeomorphisms from which we *can* infer a subgraph homeomorphism from S to T.

When we apply contraction to replace a series of edges E separated by contractable vertices by a single edge e with the intermediate vertices contracted, we save the label sets for each intermediate vertex in E in the order that they appear (following the direction of e) before starting the search algorithm.

During edge-path matching in the partial mapping search, we may need to map an edge (u, v) that contains contracted vertices. For each edge $\in E_{cont}$ with the same source- and target vertex we recall their lists of contracted label sets. Each time we find a path candidate for (u, v) we check compatibility of that path with each of those lists of label sets.



(a) Part of a source graph that may be contracted



Edge	Contracted
(s_1, s_3)	{[∅]}

(b) The result of contraction

Figure 6.1: Contraction applied to vertex s_2, a vertex with indegree 1 and outdegree 1. During contraction, we save that a single vertex has been contracted with an empty label set.

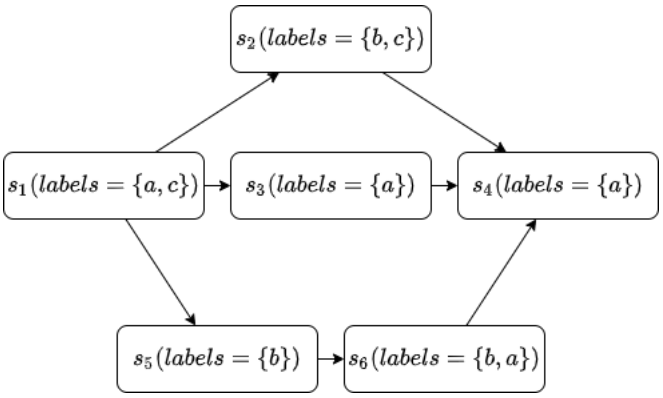
We then retrieve all other paths already found for edges with this source- and target and use an all-different constraint (see Section 7.2.2) to verify that each edge $\in E_{cont}$ has a compatible path. Whenever the number of paths is smaller than the number of edges (i.e. paths still need to be found futher in the partial mapping search), we assume paths that are compatible with every list of label sets.

If the all-different constraint fails, it implies that from the current partial mapping we cannot find paths that emulate the contracted vertices, and we backtrack. If the all-different constraint succeeds, finding a subgraph homeomorphism is still possible and we continue.

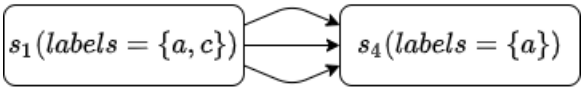
This optimisation introduces some delay since the algorithm needs to check compatibility between label set sequences and paths and solve an all-different constraint. However, it also saves time since it reduces the source graph's size.

Contraction for undirected graphs

Although we are focued on directed graphs in our algorithm, it is interesting to note that each optimisation is also applicable to undirected graphs, including contraction. Contraction can be applied to undirected graphs by temporarily applying a uniform direction to each chain of 2-degree vertices in the source graph and then applying directed contraction, after which the edges may lose their direction again. Then, during the compatibility checking process we perform the same procedure, making sure the list of label sets is checked in the correct direction, i.e. for each temporarily directed edge $(u, v) \in E_{cont}$ and partial mapping M, the path $M(u, v)$ in T should be checked in the direction from $M(u)$ to $M(v)$.



(a) A source graph before contraction



Edge	Contracted
(s_1, s_4)	$\{[\{b, c\}], [\{a\}], [\{b\}, \{b, a\}]\}$

(b) The same source graph after contraction.

Figure 6.2: Contraction applied to a source graph with labels. With this optimisation, 3 fewer vertices and 4 fewer edges need to be mapped. During contraction, we save the label sets of the contracted vertices in the order of the contracted path.

6.7 Avoiding unintended current flow

As mentioned in Chapter 5, the concrete FPGA may contain transistors that are always enabled, i.e. cannot be configured to block the flow of electrical current and function as a diode. If we would ignore this property of transistors, we may obtain a mathematically sound subgraph homeomorphism that does not include every unconfigurable transistor from the target graph in its mappings. If some unconfigurable transistor (or a sequence of them) connect two vertices in the target graph that **are** part of the mapping, then the subgraph homeomorphism does not describe a valid emulation: if the state of two components are independent in the virtual FPGA they should be independent in the concrete FPGA as well to preserve the semantics of the FPGA configuration. An example of this effect is given in Figure 6.3. To avoid this issue, we propose two measures:

6.7.1 Avoiding unintended current flow from/to paths

Our first remedy is to avoid adding target graph vertices to the partial mapping that are connected to some path already present in the edge-path partial mapping through unconfigurable vertices.

These vertices can get electrical currents whenever it flows though the path, possibly breaking the program's semantics. Therefore, we avoid adding these vertices to the partial mapping by deleting them from the graph in search branches that include the corresponding paths. Deleting them prevents us from adding them to the partial mapping and having our semantics changed.

More formally:

Let *unconfigurable* $\in (V_T \times V_T)$ be the relation between two vertices if they are connected

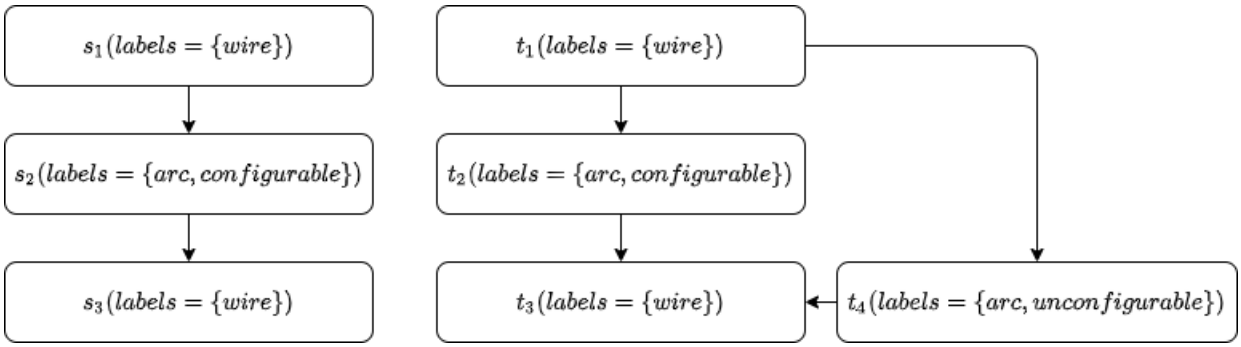


Figure 6.3: An example of a pair of FPGAs with an unconfigurable transistor in the concrete FPGA. There exists a mathematically sound subgraph homeomorphism $(\{s_1 \to t_1, s_2 \to t_2, s_3 \to t_3\}, \{(s_1, s_2) \to [(t_1, t_2)], (s_2, s_3) \to [(t_2, t_3)]\})$. However, a configuration in which the virtual FPGA transistor s_2 is disabled cannot be translated to a configuration in the target FPGA with the same semantics. While t_2 can be configured to be disabled, the electrical current can always flow from t_1 to t_3 through t_4. The algorithm should avoid finding subgraph homeomorphisms like these.

through a vertex representing a unconfigurable transistor, i.e:

$$\textit{unconfigurable} := \{(t_1, t_2) \in (V_T \times V_T) | \exists u \in (V_T \setminus \{t_1, t_2\}). \quad \begin{array}{l} \texttt{UNCONFIGURABLE} \in L(u) \wedge \\ \texttt{CONFIGURABLE} \notin L(u) \wedge \\ \{(t_1, u), (u, t_2)\} \subseteq E_T\} \end{array}$$

Let *unconfigurable** be the symmetric closure of the transitive closure of *unconfigurable*. Then, whenenever we add an edge-path pair (e, p) to the partial mapping, we delete each vertex x for which $\exists v \in$ *intermediate*$(p).(x, v) \in$ *unconfigurable**. Whenever (e, p) is later removed from the partial mapping, each such vertex and their edges is added back to the target graph. After all, these vertices *are* allowed to be in our mapping if p is not.

6.7.2 Avoiding unintended current flow between mapped vertices

Our second remedy is to avoid adding target graph vertices to the partial mapping that are connected to some target graph vertex already present in the vertex-vertex partial mapping through unconfigurable vertices. Allowing this could result in wires in the concrete FPGA receiving an electrical current that is not part of the program's semantics.

We also do this by deleting vertices: this time, whenever we add a vertex-vertex pair (s, t) to the partial mapping, we delete each vertex x for which $(x, t) \in$ *unconfigurable**. Whenever (s, t) is later removed from the partial mapping, each such vertex and their edges is added back to the target graph.

Algorithm 3: ndSHD2-ext

Inputs: the source graph G_S and the target graph G_T
Outputs: *found*, indicating whether a valid subgraph homeomorphism has been found.
if *contraction* **then**
    $G_S, chains \longleftarrow contract(G_S)$ `// sec. 6.6.3`
end
return ndSHD2-ext(*vmap*, *emap*)`// 4`;

Algorithm 4: ndSHD2-ext

```
1  Inputs: the current vertex-vertex partial mapping vmap, the current edge-path emap partial mapping
2  Outputs: found, indicating whether a valid subgraph homeomorphism has been found.
3  if s is complete then
4  |  return true;
5  end
6  found ⟵ false
7  while !found ∧ ∃ valid node/edge-path mapping pair do
8  |  if hasUnmatchedEdge() then
9  |  |  (e_S, p_T) ⟵ GetNextEdgePathPair();
10 |  |  if isUnnecessarilyLong(p_T)∨ // sec. 6.6.1
   |  |   wouldPrune-ext(vmap, emap ∪ {(e_S, p_T)})∨// See Algorithm 5
   |  |   (contraction ∧ ¬chainsCompatible(emap, from(e_S), to(e_S), p_T))∨// sec. 6.6.3
   |  |   unconfigurableCover(intermediate(p_T)) ∩ (emap ∪ vmap) ≠ ∅ then // sec. 6.7.1
11 |  |  |  continue;
12 |  |  end
13 |  |  V_T ⟵ V_T \ unconfigurableCover(intermediate(p_T))// sec. 6.7.1;
14 |  |  emap ⟵ emap ∪ {(e_S, p_T)};
15 |  |  found ⟵ ndSHD2-ext(vmap, emap);
16 |  |  if found then
17 |  |  |  return true;
18 |  |  end
19 |  |  emap ⟵ emap ∩ {(e_S, p_T)};
20 |  |  V_T ⟵ V_T ∪ unconfigurableCover(intermediate(p_T))// sec. 6.7.1;
21 |  end
22 |  else
23 |  |  (v_S, v_T) ⟵ GetNextNodePair();
24 |  |  if wouldPrune-ext(vmap ∪ {(v_S, v_T)}, emap)∨// See Algorithm 5
   |  |   unconfigurableCover(v_T) ∩ (vmap ∪ emap) ≠ ∅ then // sec. 6.7.2
25 |  |  |  continue;
26 |  |  end
27 |  |  V_T ⟵ V_T \ unconfigurableCover(v_T)// sec. 6.7.2;
28 |  |  vmap ⟵ vmap ∪ {(v_S, v_T)};
29 |  |  found ⟵ ndSHD2-ext(vmap, emap);
30 |  |  if found then
31 |  |  |  return true;
32 |  |  end
33 |  |  vmap ⟵ vmap ∩ {(v_S, v_T)};
34 |  |  V_T ⟵ V_T ∪ unconfigurableCover(v_T)// sec. 6.7.2;
35 |  end
36 end
37 return false;
```

Algorithm 5: wouldPrune-ext

Inputs: the current vertex-vertex partial mapping *vmap*, the current edge-path *emap* partial mapping
Outputs: whether pruning should be applied.
if $\mathit{serialPruning} \vee (\mathit{parallelPruning} \wedge \mathit{pruningThreadFailed})$ **then**
    $\mathit{domains} \longleftarrow \mathit{filterDomains}(\mathit{vmap}, \mathit{emap})$;
end
else if *cachedPruning* **then**
    $\mathit{domains} \longleftarrow \mathit{updateDomains}(\mathit{vmap}, \mathit{emap})$;
end
else
    return false;
end
$\mathit{toPrune} \longleftarrow$ false;
if *ZeroDomain* **then**
    $\mathit{toPrune} \longleftarrow \emptyset \in \mathit{values}(\mathit{domains})$;
end
else if *AllDifferent* **then**
    $\mathit{toPrune} \longleftarrow \neg \mathit{satisfiesAllDifferent}(\mathit{domains})$;
end
if $\mathit{parallelPruning} \wedge \neg \mathit{toPrune}$ **then**
    $\mathit{pruningThreadFailed} \longleftarrow$ false;
end
return *toPrune*;

Algorithm 6: chainsCompatible

Inputs: the current edge-path *emap* partial mapping, the source vertex v_s and the target vertex v_t of the edge to be added, the path to be associated with that edge p
Outputs: Whether we need are safe to continue (true) or need to backtrack (false) due to contraction compatibility issues.
$\mathit{labelsetSequences} \longleftarrow \mathit{chains}(v_s, v_t)$;
$\mathit{pathBag} \longleftarrow \{\mathit{emap}(e).e \in E_S \wedge \mathit{from}(e) = v_s \wedge \mathit{to} = v_t\} \cup \{p\}$;
$\mathit{edgesToGo} \longleftarrow |\{e \in E_S.e \notin \mathit{emap} \wedge \mathit{from}(e) = v_s \wedge \mathit{to}(e) = v_t\}|$
repeat *edgesToGo* **times**
    $\mathit{pathBag} \longleftarrow \mathit{pathBag} \cup \{\top\}$;
end
for $\mathit{seq} \in \mathit{labelsetSequences}$ **do**
    $\mathit{domain}(\mathit{seq}) \longleftarrow \{p' \in \mathit{pathBag}.\mathit{containsSubSequence}(\mathit{seq}, p')\}$;
end
return $\mathit{satisfiesAllDifferent}(\mathit{domains})$;

Algorithm 7: containsSubSequence

Inputs: A sequence of label sets $S_0 \dots S_m$, a path $v_0 \dots v_n$
Outputs: Whether the path $v_0 \dots v_n$ is compatible with the label set sequence $S_0 \dots S_m$, i.e. is able to emulate it.
if $v_0 \dots v_n = \top \vee S_0 \dots S_m = []$ **then**
    return true;
end
else if $v_0 \dots v_n = []$ **then**
    return false;
end
else if $S_0 \subseteq L_T(v_0)$ **then**
    return *containsSubSequence*$(S_1 \dots S_n, v_1 \dots v_n)$;
end
else
    return *containsSubSequence*$(S_0 \dots S_n, v_1 \dots v_n)$;
end

7 PRUNING

During the search for a complete matching, during which the algorithm only has *partial* matchings, the algorithm will often explore dead branches, i.e. branches of the search tree that will not eventually lead to a homeomorphism. Exploring an entire dead branch may be costly: its size is exponential[1] 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 efficiently[2]. 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. Xiao's algorithm uses cached zero-domain N-reachability, which we will implement for our algorithm. We will also implement diffent methods, each with different tradeoffs. We implement these algorithms without precomputed paths, i.e. we run a pathfinding algorithm during the partial mapping search instead and optionally cache the results.

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. See Figure 7.1 for an example. 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 7.1). 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 7.2.

7.1 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

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

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

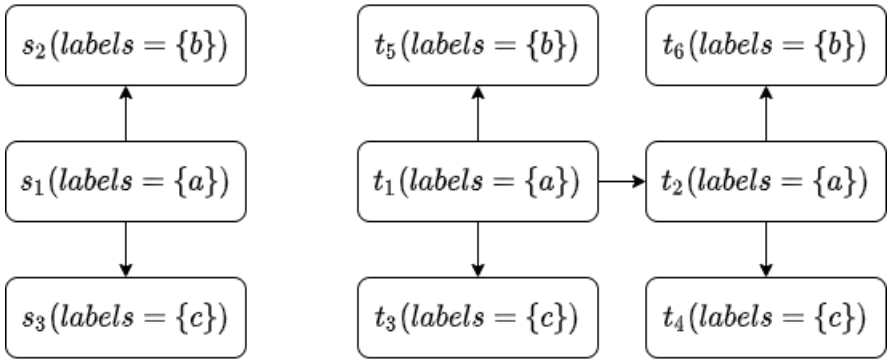


Figure 7.1: An example case. With an empty partial mapping, the domain of s_1 is $\{t_1, t_2\}$, every compatible vertex in the target graph. The domain of s_2 is initially $\{t_5, t_6\}$ and the domain of s_3 $\{t_3, t_4\}$. This holds for any filtering method mentioned in this chapter. When pairs are added to the partial mapping, the domain of each source graph vertex may be reduced.

homeomorphism that can be found from this search branch with vertex-on-vertex matching M_V it holds that $\forall(s \to t) \in M_V.t \in \textit{domain}(s)$.

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). While technically any combination of these methods can be in combination (i.e. by intersecting the domains they find), we limit ourselves with the assumption that each method is combined with every computationally cheaper filtering method.

7.1.1 Labels and neighbours

The weakest and fastest method to obtain domains for some source graph vertex u is by selecting each target graph vertex v that is not already used in the current partial mapping, has compatible labels and has compatible in- and outdegrees:

Definition 7.1.1 (compability under label constraint)**.** If M is the current partial matching, then:

$$\begin{aligned} \textit{compatible}_{\textit{LABEL}}(s,t) := \quad & t \notin M && \wedge \\ & L(s) \subseteq L(t) && \wedge \\ & |\textit{pred}(t)| \geq |\textit{pred}(s)| && \wedge \\ & |\textit{succ}(t)| \geq |\textit{succ}(s)| \end{aligned}$$

This method satisfies criterium 1 since every homeomorphism needs each source graph vertex s to be matched with a target graph vertex t that has at least the same label set as s, and the possibility of connecting with each mapped predecessor and successor of s. It has a low average computational complexity per source-target pair ($O(|L_S|)$).

7.1.2 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 s to other unmatched source graph vertices to the in- and outdegree of the target graph vertex t 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 7.1.2 (compatibility under free neighbours constraint)**.** If M is the current partial matching, then:

$$\begin{aligned} \textit{compatible}_{\textit{FN}}(s,t) := \quad & \textit{compatible}_{\textit{LABEL}}(s,t) && \wedge \\ & |\textit{pred}(t) \setminus M| \geq |\textit{pred}(s) \setminus M| && \wedge \\ & |\textit{succ}(t) \setminus M| \geq |\textit{succ}(s) \setminus M| \end{aligned}$$

This method satisfies criterium 1 since each unmatched neighbour must yet be matched with a target graph vertex that is (vertex disjointly) connected with the target graph vertex. For this purpose, unused connection to the vertex are required. It has a slightly higher average computational complexity per source-target pair since (if no cached approach is chosen) each neighbour of s and of t needs to be counted: $O(|L_S|+|E_S|/|V_S|+|E_T|/|V_T|)$.

7.1.3 Reachability of matched vertices (M-filtering)

A strong (and computationally very expensive) method of filtering the domain we will use to calculate whether a target vertex t is in the domain of a source graph vertex s in addition to label- indegree and outdegree compability is to check that t can reach the mapped vertices of successors of s and that t can be reached from the mapped vertices of predecessors of s, 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 7.2. Formally:

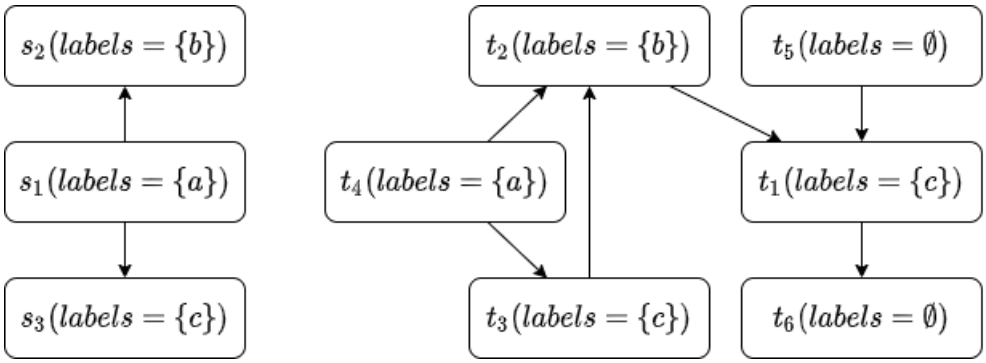


Figure 7.2: After vertex placements $s_1 \rightarrow t_4$ and $s_2 \rightarrow t_2$, the domain for vertex s_3 would normally be $\{t_1, t_3\}$. However, by checking for reachability from t_4 we can reduce this to $\{t_3\}$. The numbers represent the matching order and the circle styles represent labels.

Definition 7.1.3 (compatibility under reachability constraint)**.** If M is the current partial mapping and P is the set of paths in the target graph, then:

$$\begin{aligned} \textit{compatible}_{M-REACH}(s,t) := & \textit{compatible}_{FN}(s,t) \quad \wedge \\ & \textit{compatible}_{M-REACH,pred}(s,t) \wedge \\ & \textit{compatible}_{M-REACH,succ}(s,t) \end{aligned}$$

where:

$$\begin{aligned} \textit{compatible}_{M-REACH,pred}(s,t) := \forall s' \in (\textit{pred}(s) \cap M).\exists p \in P. \quad & \textit{first}(p) = M(s') \quad \wedge \\ & \textit{last}(p) = t \quad \wedge \\ & \textit{intermediate}(p) \cap M = \emptyset \\ \textit{compatible}_{M-REACH,succ}(s,t) := \forall s' \in (\textit{succ}(s) \cap M).\exists p \in P. \quad & \textit{first}(p) = t \quad \wedge \\ & \textit{last}(p) = M(s') \quad \wedge \\ & \textit{intermediate}(p) \cap M = \emptyset \end{aligned}$$

If Dijkstra's algorithm is used to find p, we have a complexity of $O(|E_T| + |V_T| * \textit{log}(|V_T|))$ for each neighbour, resulting in a total average case complexity per source-target pair of $O(|L_S| + (|E_T| + |V_T| * \textit{log}(|V_T|)) * |E_S|/|V_S|)$

7.1.4 Reachability of neighbourhood (N-filtering)

In addition to filtering domains based on reachability from- and to matched vertices that have a domain of one target graph vertex, we can also perform this on unmatched vertices that have multiple vertices in their domain. In this case, any single vertex in the domain of the source graph neighbour qualifies. 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.

Since this process is dependent on the domain of other vertices and reduces the size of domains itself, this process yields new (smaller) domains and is therefore repeated until a fixed point is reached.

Definition 7.1.4 (compatibility under neighbourhood reachability constraint)**.** If M is the current partial mapping and P is the set of paths in the target graph, then neighbourhood reachability compatibility is defined as

$$\mathit{compatible}_{N-REACH}(s,t) := \lim_{i \to +\infty} \mathit{nreach}_i(s,t)$$

where:

$$\mathit{nreach}_i(s,t) = \begin{cases} \mathit{compatible}_{M-REACH}(s,t) & i = 0 \\ \mathit{ncomp}_{i,pred}(s,t) \wedge \mathit{ncomp}_{i,succ}(s,t) & \text{otherwise} \end{cases}$$

$$\begin{aligned} \mathit{ncomp}_{i,pred}(s,t) := \quad & \forall s' \in \mathit{pred}(s). \exists t' \in V_T. \mathit{nreach}_{i-1}(s',t') \wedge \\ & \exists p \in P. \quad \mathit{first}(p) = t' \wedge \\ & \qquad\quad \mathit{last}(p) = t \wedge \\ & \qquad\quad \mathit{intermediate}(p) \cap M = \emptyset \\ \mathit{ncomp}_{i,succ}(s,t) := \quad & \forall s' \in \mathit{succ}(s). \exists t' \in V_T. \mathit{nreach}_{i-1}(s',t') \wedge \\ & \exists p \in P. \quad \mathit{first}(p) = t \wedge \\ & \qquad\quad \mathit{last}(p) = t' \wedge \\ & \qquad\quad \mathit{intermediate}(p) \cap M = \emptyset \end{aligned}$$

7.2 Pruning methods

7.2.1 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. Since each source graph vertex must be matched with some target graph vertex in a subgraph homeomorphism, this implies the search is in a dead branch. 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 7.1.

7.2.2 AllDifferent pruning

The AllDifferent constraint specifies that given some assignment of domains to variables, each variable should have a non-empty domain (i.e. AllDifferent is stronger than ZeroDomain) *and* and some injective mapping exists from variables to values in their do-

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 7.1: 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.

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 7.2: 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.

mains. Since a homeomorphism requires the vertex-vertex mapping to be injective (and extending a non-injective vertex-vertex mapping can never make it injective again), the AllDifferent pruning algorithm backtracks whenever no such injective mapping exists. Alldifferent uses quadratic space since each domain needs to be known at the same time (in contrast with zero-domain, in which the knowledge of each domain separately is sufficient). An example of a domain assignment where AllDifferent prunes the search space but ZeroDomain does not is shown in Table 7.2.

7.3 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.

7.3.1 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.

7.3.2 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 space[3] (for each source graph vertex $\in |V_{source}|$ it needs to store a domain of average size $O(|V_{target}|)$). If M-reachability filtering or N-reachability filtering 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.

7.3.3 Parallel calculation

Finally, we can perform the domain filtering and procedure in a seperate 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, since that requires updating the cache *every* time the partial matching is extended - and performance benefits are only gained when the partial matching is only occasionally queried.

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

8 END-TO-END EXAMPLE

To illustrate the process documented in this thesis, we provide an end-to-end example on a small use case. This entails the process from modeling FPGAs all the way up to applying an emulation mapping on configurations of the virtual FPGA.

8.1 FPGAs

We introduce the virtual FPGA *VirSample*. This simple FPGA has two inputs, has a 2-input-1-ouput LUT and two outputs (one of which is always zero). Figure **??** shows the layout and different configurations of this virtual FPGA.

Suppose we want to emulate configurations of VirSample on a real, physical FPGA ConcreteSample (shown in Figure 8.2). This FPGA has a 2-input-2-output logic cell, a register and a multiplexer (a component that selects one of its two input sets as output depending on its configuration). Intuitively, we may find an emulation mapping by hand:

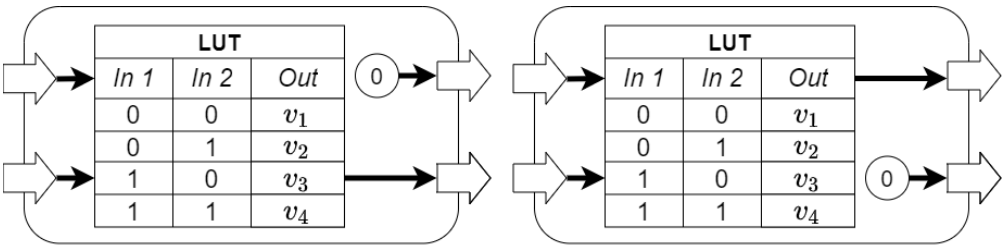


Figure 8.1: Both configurations of VirSample. The left configuration is specified by configuration bit $v_5 = 0$ while the right configuration is specified by $v_5 = 1$.

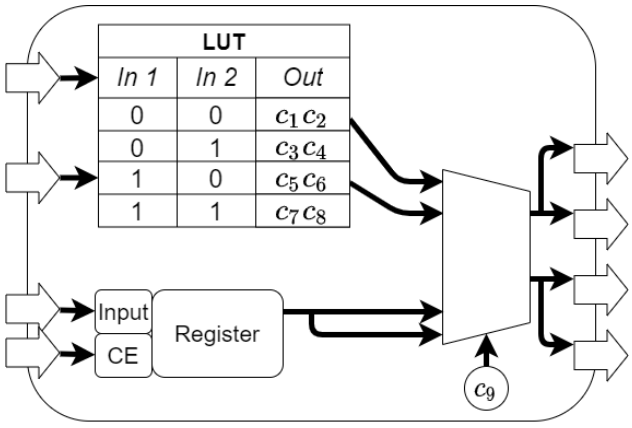


Figure 8.2: The ConcreteSample FPGA. It has 9 configurable bits, 8 of which in its lookup table and one of which a selector of its multiplexer.

- $c_1 \longleftarrow v_5 \wedge v_1$
- $c_2 \longleftarrow \neg v_5 \wedge v_1$
- $c_3 \longleftarrow v_5 \wedge v_2$
- $c_4 \longleftarrow \neg v_5 \wedge v_1$
- $c_5 \longleftarrow v_5 \wedge v_3$
- $c_6 \longleftarrow \neg v_5 \wedge v_3$
- $c_7 \longleftarrow v_5 \wedge v_4$
- $c_8 \longleftarrow \neg v_5 \wedge v_4$
- $c_9 \longleftarrow 0$
- virtual input 1 $=$ concrete input 1
- virtual input 2 $=$ concrete input 2
- virtual output 1 $=$ concrete output 1
- virtual output 2 $=$ concrete output 3

We will show that a similar emulation mapping will also be the result of applying our method. First, we will design graph models for both FPGAs in Section 8.2. We will find a subgraph homeomorphism between those graphs in Section 8.3.1. Finally, we will retrieve the emulation mapping in Section 8.4.

8.2 Graph models

We model our FPGAs using the model specified in Chapter 5, with one addition: we model LUTs with a new label `LOGIC`. Since we only model physical structures (i.e. wires, transistors and logical components) instead of semantics, we need to think of ways that VirSample could have been implemented using these physical structures. After all, this FPGA does not physically exist. Some implementations of VirSample will yield graph models that have subgraph homeomorphism embeddings in ConcreteSample's graph while some will not. In this example, we will make a single compact graph model, although for practi-

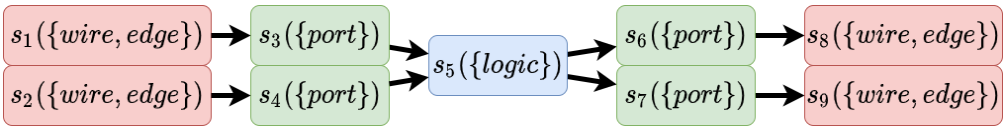


Figure 8.3: Graph model of the virtual FPGA.

cal use we recommend trying out different models. This graph model is shown in Figure 8.3. This model assumes that the physical implementation of VirSample uses a 2-input-2-output LUT, with the limitation that for each input combination in the LUT, at least one output should be configured to be zero, depending on v_5.

Obtaining the graph model for the concrete FPGA is easier: since ConcreteSample physically exists, we already know where transistors, wires and LUTs are used. Using the model from Chapter 5 with the LUT addition we obtain the graph shown in Figure 8.4. We omit the register since we logically induce that, since VirSample only uses combinatorial components, an emulation on ConcreteSample would never include usage of its register.

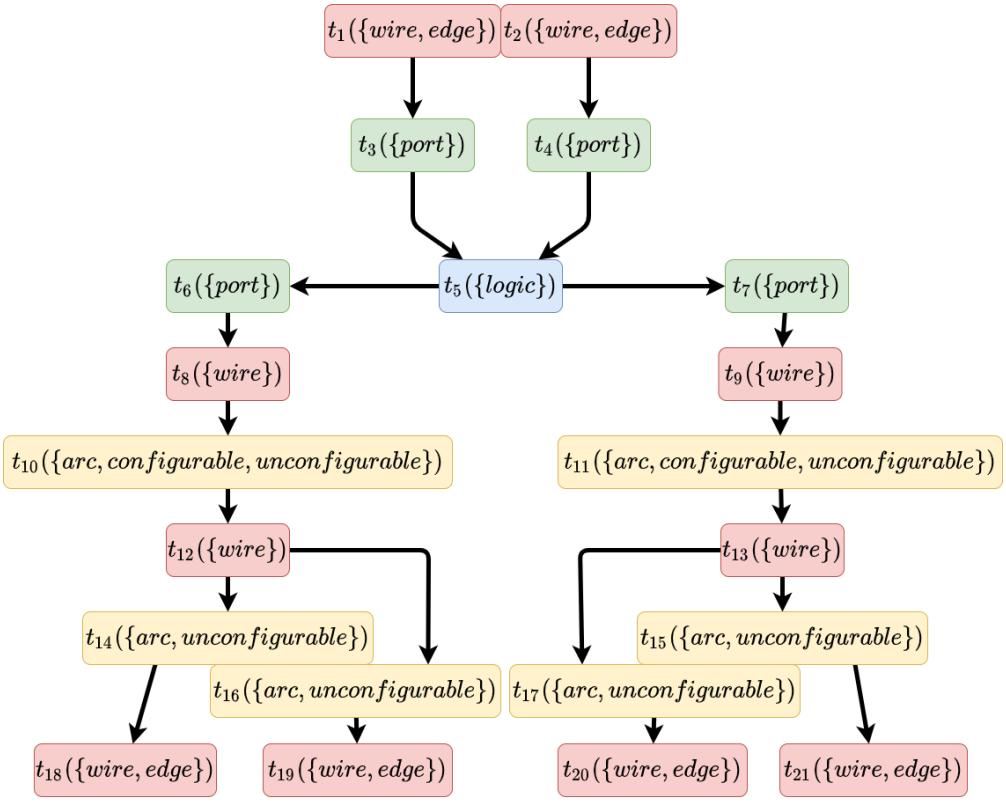


Figure 8.4: Graph model of the concrete FPGA.

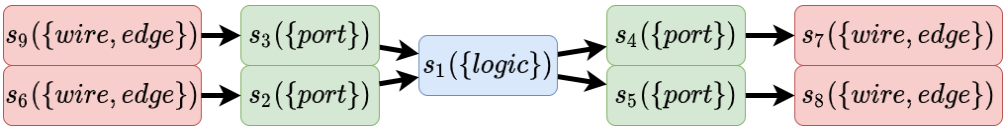


Figure 8.5: Graph model of the virtual FPGA with vertices ordered using GreatestConstrainedFirst.

8.3 Finding a subgraph homeomorphism

8.3.1 Applying ordering

We apply GreatestConstrainedFirst to our source graph from Figure 8.3 to obtain the graph shown in Figure 8.5.

Furthermore, we order the target graph vertices from the target graph from Figure 8.4 by degree to obtain the graph shown in Figure 8.5

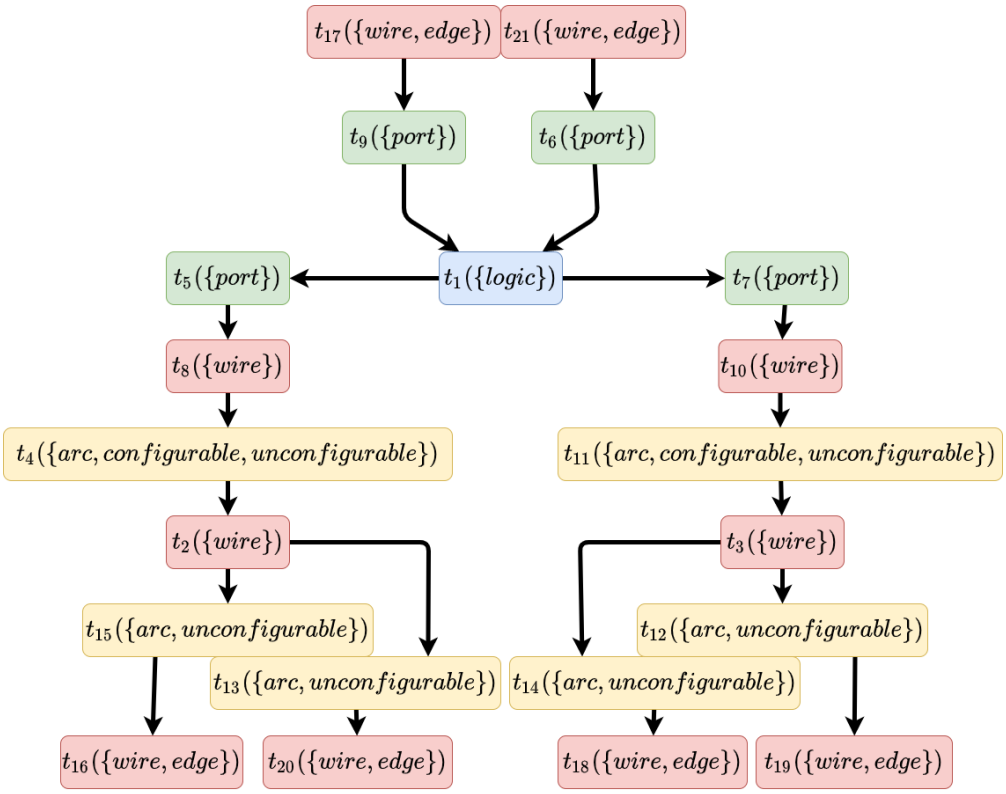
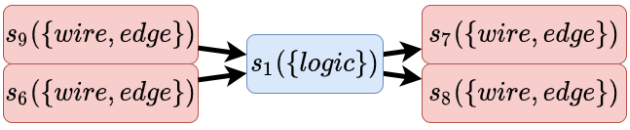


Figure 8.6: Graph model of the concrete FPGA with vertices ordered by degree.

8.3.2 Applying contraction

We use our algorithm with contraction enabled, meaning we will preprocess our source graph to reduce its size, keeping track of contracted vertices to use later in the algorithm. Contracting each source graph vertex with indegree 1 and outdegree 1 leaves us with the graph shown in Figure 8.7.



Edge	Contracted
(s_9, s_1)	$\{[\{port\}]\}$
(s_6, s_1)	$\{[\{port\}]\}$
(s_1, s_7)	$\{[\{port\}]\}$
(s_1, s_8)	$\{[\{port\}]\}$

Figure 8.7: Contracted graph model of the virtual FPGA.

8.3.3 Running algorithm

We start with a partial mapping $(\textit{vmap}, \textit{emap})$ where *vmap* and *emap* are both empty and run Algorithm 4.

- (lines 8, 23) There exists no unmatched edge, so retrieve the next node pair (s_1, t_1).
- (line 24) The pruner (which uses N-reachability filtering) obtains the domains:
  $s_1 \to \{t_1\}$
  $s_6 \to \{t_{17}, t_{21}\}$
  $s_7 \to \{t_{16}, t_{18}, t_{19}, t_{20}\}$
  $s_8 \to \{t_{16}, t_{18}, t_{19}, t_{20}\}$
  $s_9 \to \{t_{17}, t_{21}\}$
  Since an injective mapping exists from this domain, we do not prune.
- (line 28) We add (s_1, t_1) to *vmap*, which is now $\{(s_1, t_1)\}$.
- (line 29) We enter a recursive call.
- (lines 8, 23) There exists no unmatched edge, so retrieve the next node pair. For briefness, we will omit attempts to match s_6 with $t_1 \ldots t_{16}$: these will each result in being pruned in line 11 because of reachability and label compatibility problems, until we try (s_6, t_{17}).

- (line 24) The pruner (which uses N-reachability filtering) obtains the domains:
  $s_1 \to \{t_1\}$
  $s_6 \to \{t_{17}\}$
  $s_7 \to \{t_{16}, t_{18}, t_{19}, t_{20}\}$
  $s_8 \to \{t_{16}, t_{18}, t_{19}, t_{20}\}$
  $s_9 \to \{t_{21}\}$
  Since an injective mapping exists from this domain, we do not prune.
- (line 28) We add (s_6, t_{17}) to ***vmap***, which is now $\{(s_1, t_1), (s_6, t_{17}))\}$.
- (line 29) We enter a recursive call.
- (line 8) There exists an unmatched edge (s_6, s_1), so retrieve the next edge-path pair (using DFS path iteration) $(s_6 \to s_1, t_{17} \to t_9 \to t_1)$.
- (line 10) Since no shortcuts exist in the path $t_{17} \to t_9 \to t_1$, ***isUnnecessarilyLong***$(t_{17} \to t_9 \to t_1)$ is false.
- (line 10) The pruner (which uses N-reachability filtering) obtains the domains:
  $s_1 \to \{t_1\}$
  $s_6 \to \{t_{17}\}$
  $s_7 \to \{t_{16}, t_{18}, t_{19}, t_{20}\}$
  $s_8 \to \{t_{16}, t_{18}, t_{19}, t_{20}\}$
  $s_9 \to \{t_{21}\}$
  Since an injective mapping exists from this domain, we do not prune.
- (line 10) Contraction is enabled, and we find that the path $t_{17} \to t_9 \to t_1$ satisfies the requirements of having a single intermediate vertex with at least the label `PORT` (vertex t_9). Therefore, we do not prune.
- (line 14) We add $(s_6 \to s_1, t_{17} \to t_9 \to t_1)$ to ***emap***, which is now $\{(s_6 \to s_1, t_{17} \to t_9 \to t_1)\}$.
- (line 15) We enter a recursive call.
- (lines 8, 23) There exists no unmatched edge, so retrieve the next node pair (s_7, t_{16}). For briefness, we will omit attempts to match s_7 with $t_1 \ldots t_{15}$: these will each result in being pruned in line 11 because of reachability and label compatibility problems, until we try (s_7, t_{16}).
- (line 24) The pruner (which uses N-reachability filtering) obtains the domains:
  $s_1 \to \{t_1\}$
  $s_6 \to \{t_{17}\}$
  $s_7 \to \{t_{16}\}$
  $s_8 \to \{t_{18}, t_{19}, t_{20}\}$
  $s_9 \to \{t_{21}\}$
  Since an injective mapping exists from this domain, we do not prune.
- (line 28) We add (s_7, t_{16}) to ***vmap***, which is now $\{(s_1, t_1), (s_6, t_{17}), (s_7, t_{16}))\}$.
- (line 29) We enter a recursive call.
- (line 8) There exists an unmatched edge (s_1, s_7), so retrieve the next edge-path pair (using DFS path iteration) $(s_1 \to s_7, t_1 \to t_5 \to t_8 \to t_4 \to t_2 \to t_{15} \to t_{16})$.
- (line 10) Since no shortcuts exist in the path $t_1 \to t_5 \to t_8 \to t_4 \to t_2 \to t_{15} \to t_{16}$, ***isUnnecessarilyLong***$(t_1 \to t_5 \to t_8 \to t_4 \to t_2 \to t_{15} \to t_{16})$ is false.

- (line 10) The pruner (which uses N-reachability filtering) obtains the domains:
  $s_1 \to \{t_1\}$
  $s_6 \to \{t_{17}\}$
  $s_7 \to \{t_{16}\}$
  $s_8 \to \{t_{18}, t_{19}\}$
  $s_9 \to \{t_{21}\}$
  Since an injective mapping exists from this domain, we do not prune.
- (line 10) Contraction is enabled, and we find that the path $t_1 \to t_5 \to t_8 \to t_4 \to t_2 \to t_{15} \to t_{16}$ satisfies the requirements of having a single intermediate vertex with at least the label `PORT` (vertex t_5). Therefore, we do not prune.
- (line 13) Since vertex t_{20} is in the cover reachable by vertices from this path through unconfigurable transistors, we delete it from the graph until this path is backtracked.
- (line 14) We add $(s_1 \to s_7, t_1 \to t_5 \to t_8 \to t_4 \to t_2 \to t_{15} \to t_{16})$ to ***emap***, which is now $\{(s_6 \to s_1, t_{17} \to t_9 \to t_1), (s_1 \to s_7, t_1 \to t_5 \to t_8 \to t_4 \to t_2 \to t_{15} \to t_{16})\}$.
- (line 15) We enter a recursive call.
- (lines 8, 23) There exists no unmatched edge, so retrieve the next node pair (s_8, t_{18}). For briefness, we will omit attempts to match s_7 with $t_1 \ldots t_{17}$: these will each result in being pruned in line 11 because of reachability and label compatibility problems, until we try (s_8, t_{18}).
- (line 24) The pruner (which uses N-reachability filtering) obtains the domains:
  $s_1 \to \{t_1\}$
  $s_6 \to \{t_{17}\}$
  $s_7 \to \{t_{16}\}$
  $s_8 \to \{t_{18}\}$
  $s_9 \to \{t_{21}\}$
  Since an injective mapping exists from this domain, we do not prune.
- (line 28) We add (s_8, t_{18}) to ***vmap***, which is now $\{(s_1, t_1), (s_6, t_{17}), (s_7, t_{16}), (s_8, t_{18})\}$.
- (line 29) We enter a recursive call.
- (line 8) There exists an unmatched edge (s_1, s_8), so retrieve the next edge-path pair (using DFS path iteration) $(s_1 \to s_8, t_1 \to t_7 \to t_{10} \to t_{11} \to t_3 \to t_{14} \to t_{18})$.
- (line 10) Since no shortcuts exist in the path $t_1 \to t_7 \to t_{10} \to t_{11} \to t_3 \to t_{14} \to t_{18}$, ***isUnnecessarilyLong***$(t_1 \to t_7 \to t_{10} \to t_{11} \to t_3 \to t_{14} \to t_{18})$ is false.
- (line 10) The pruner (which uses N-reachability filtering) obtains the domains:
  $s_1 \to \{t_1\}$
  $s_6 \to \{t_{17}\}$
  $s_7 \to \{t_{16}\}$
  $s_8 \to \{t_{18}\}$
  $s_9 \to \{t_{21}\}$
  Since an injective mapping exists from this domain, we do not prune.
- (line 10) Contraction is enabled, and we find that the path $t_1 \to t_7 \to t_{10} \to t_{11} \to t_3 \to t_{14} \to t_{18}$ satisfies the requirements of having a single intermediate vertex with at least the label `PORT` (vertex t_7). Therefore, we do not prune.

- (line 13) Since vertex t_{19} is in the cover reachable by vertices from this path through unconfigurable transistors, we delete it from the graph until this path is backtracked.
- (line 14) We add $(s_1 \to s_7, t_1 \to t_5 \to t_8 \to t_4 \to t_2 \to t_{15} \to t_{16})$ to ***emap***, which is now:
  $\{(s_6 \to s_1, t_{17} \to t_9 \to t_1),$
  $(s_1 \to s_7, t_1 \to t_5 \to t_8 \to t_4 \to t_2 \to t_{15} \to t_{16}),$
  $(s_1 \to s_8, t_1 \to t_7 \to t_{10} \to t_{11} \to t_3 \to t_{14} \to t_{18})\}$.
- (line 15) We enter a recursive call.
- (lines 8, 23) There exists no unmatched edge, so retrieve the next node pair. For briefness, we will omit attempts to match s_6 with $t_1 \dots t_{20}$: these will each result in being pruned in line 11 because of reachability and label compatibility problems, until we try (s_9, t_{21}).
- (line 24) The pruner (which uses N-reachability filtering) obtains the domains:
  $s_1 \to \{t_1\}$
  $s_6 \to \{t_{17}\}$
  $s_7 \to \{t_{16}\}$
  $s_8 \to \{t_{18}\}$
  $s_9 \to \{t_{21}\}$
  Since an injective mapping exists from this domain, we do not prune.
- (line 28) We add (s_9, t_{21}) to ***vmap***, which is now $\{(s_1, t_1), (s_6, t_{17}), (s_7, t_{16}), (s_8, t_{18}), (s_9, t_{21})\}$.
- (line 29) We enter a recursive call.
- (line 8) There exists an unmatched edge (s_9, s_1), so retrieve the next edge-path pair (using DFS path iteration) $(s_9 \to s_1, t_{21} \to t_6 \to t_1)$.
- (line 10) Since no shortcuts exist in the path $t_{21} \to t_6 \to t_1$, ***isUnnecessarilyLong***$(t_{21} \to t_6 \to t_1)$ is false.
- (line 10) The pruner (which uses N-reachability filtering) obtains the domains:
  $s_1 \to \{t_1\}$
  $s_6 \to \{t_{17}\}$
  $s_7 \to \{t_{16}\}$
  $s_8 \to \{t_{18}\}$
  $s_9 \to \{t_{21}\}$
  Since an injective mapping exists from this domain, we do not prune.
- (line 10) Contraction is enabled, and we find that the path $t_{21} \to t_6 \to t_1$ satisfies the requirements of having a single intermediate vertex with at least the label `PORT` (vertex t_6). Therefore, we do not prune.
- (line 14) We add $(s_9 \to s_1, t_{21} \to t_6 \to t_1)$ to ***emap***, which is now:
  $\{(s_6 \to s_1, t_{17} \to t_9 \to t_1),$
  $(s_1 \to s_7, t_1 \to t_5 \to t_8 \to t_4 \to t_2 \to t_{15} \to t_{16}),$
  $(s_1 \to s_8, t_1 \to t_7 \to t_{10} \to t_{11} \to t_3 \to t_{14} \to t_{18}),$
  $(s_9 \to s_1, t_{21} \to t_6 \to t_1)\}$.
- (line 15) We enter a recursive call.
- (line 4) Since the partial mapping is complete, we return true.
- (line 31) We return true from the recursive call, along with $(\textit{vmap}, \textit{emap})$

- (line 17) We return true from the recursive call, along with (*vmap*, *emap*)
- (line 31) We return true from the recursive call, along with (*vmap*, *emap*)
- (line 17) We return true from the recursive call, along with (*vmap*, *emap*)
- (line 31) We return true from the recursive call, along with (*vmap*, *emap*)
- (line 17) We return true from the recursive call, along with (*vmap*, *emap*)
- (line 31) We return true from the recursive call, along with (*vmap*, *emap*)
- (line 17) We return true from the recursive call, along with (*vmap*, *emap*)
- (line 31) We return true from the recursive call, along with (*vmap*, *emap*)
- (line 31) We return true from the recursive call, along with (*vmap*, *emap*)

In the end, we have a subgraph homeomorphism with the vertex-on-vertex mapping:

$\{(s_1, t_1), (s_6, t_{17}), (s_7, t_{16}), (s_8, t_{18}), (s_9, t_{21})\}$

and the following edge-on-path mapping:

$\{(s_6 \to s_1, t_{17} \to t_9 \to t_1),$
$(s_1 \to s_7, t_1 \to t_5 \to t_8 \to t_4 \to t_2 \to t_{15} \to t_{16}),$
$(s_1 \to s_8, t_1 \to t_7 \to t_{10} \to t_{11} \to t_3 \to t_{14} \to t_{18}),$
$(s_9 \to s_1, t_{21} \to t_6 \to t_1)\}$

8.4 Obtaining emulation mapping

To obtain the emulation mapping, we follow the following rules:

1. For each contracted vertex replaced by some edge e, assume that is is matched with the first label-compatible vertex in the path $M(e)$.
2. Each configurable concrete FPGA transistor that is not in the subgraph homeomorphism is configured to be disabled (i.e. does not allow an electrical current to flow through them).
3. Each configurable concrete FPGA transistor $M(s)$ that is in the subgraph homeomorphism in the vertex-vertex mapping is configured equal to transistor s.
4. Each configurable concrete FPGA transistor that is in the subgraph homeomorphism in the edge-path mapping as intermediate vertex is configured to be enabled, i.e. always letting electrical current flow through.
5. Each concrete FPGA LUT that is not in not the subgraph homeomorphism is configured to output all zeroes regardless of the input.
6. Each concrete FPGA LUT that is in the subgraph homeomorphism as part of the vertex-vertex mapping is configured based on the configuration of the virtual LUT such that if the input `PORT` vertices of the virtual FPGA graph *s-in*$_1 \dots$ *s-in*$_m$ are mapped to concrete FPGA vertices *t-in*$_1 \dots$ *t-in*$_m$ and the output `PORT` vertices of the virtual

FPGA graph *s-out*$_1$ ... *s-out*$_n$ are mapped to concrete FPGA vertices *t-out*$_1$ ... *t-out*$_n$, the function implemented by the concrete FPGA LUT from *t-in*$_1$ ... *t-in*$_m$ to *t-out*$_1$ ... *t-out*$_n$ is equal to the function implemented from *s-in*$_1$ ... *s-in*$_m$ to *s-out*$_1$ ... *s-out*$_n$. All remaining outputs should be configured to be zero.

7. Each concrete FPGA LUT that is is in the subgraph homeomorphism in the edge-path mapping as intermediate vertex is configured to output the value of its one input that is in the mapping to its one output that is in the mapping, and zeroes to each other output.
8. To emulate a signal coming into the virtual FPGA on some input wire s, one should send the same signal to the input wire $M(s)$ on the concrete FPGA.
9. The emulated output of each virtual FPGA output wire s is the output of the concrete FPGA output wire $M(s)$.

Following these rules and the obtained subgraph homeomorphism, we establish the following configuration mapping:

- $c_1 \longleftarrow v_5 \wedge v_1$
- $c_2 \longleftarrow \neg v_5 \wedge v_1$
- $c_3 \longleftarrow v_5 \wedge v_2$
- $c_4 \longleftarrow \neg v_5 \wedge v_1$
- $c_5 \longleftarrow v_5 \wedge v_3$
- $c_6 \longleftarrow \neg v_5 \wedge v_3$
- $c_7 \longleftarrow v_5 \wedge v_4$
- $c_8 \longleftarrow \neg v_5 \wedge v_4$
- $c_9 \longleftarrow 0$
- virtual input 1 $=$ concrete input 1
- virtual input 2 $=$ concrete input 2
- virtual output 1 $=$ concrete output 2
- virtual output 2 $=$ concrete output 4

Now, whenever a student creates a configuration consisting of values $v_1 \ldots v_5$ for the virtual FPGA, a configuration is also defined by the values $c_1 \ldots c_9$ for the concrete FPGA that has the same semantics.

9 BUSINESS 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 9.1. 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 9.2. 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)), bundled in 4 modules. We will focus on this type of tile to find homeomorphisms. We used Project Trellis [44] to obtain graphs of both an individual tile (shown in Figure 9.5) and of collections of adjacent tiles.

The virtual FPGA we aim to emulate on this board (which we will call *VirBoard*) is one

[1] `http://www.latticesemi.com/products/developmentboardsandkits/ecp5evaluationboard`, accessed July 10th 2020

[2] ECP5™ and ECP5-5G™ Family Data Sheet (FPGA-DS-02012 Version 1.9), accessed October 1st 2020

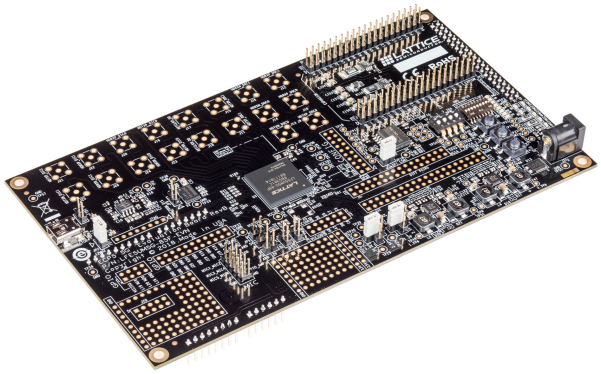
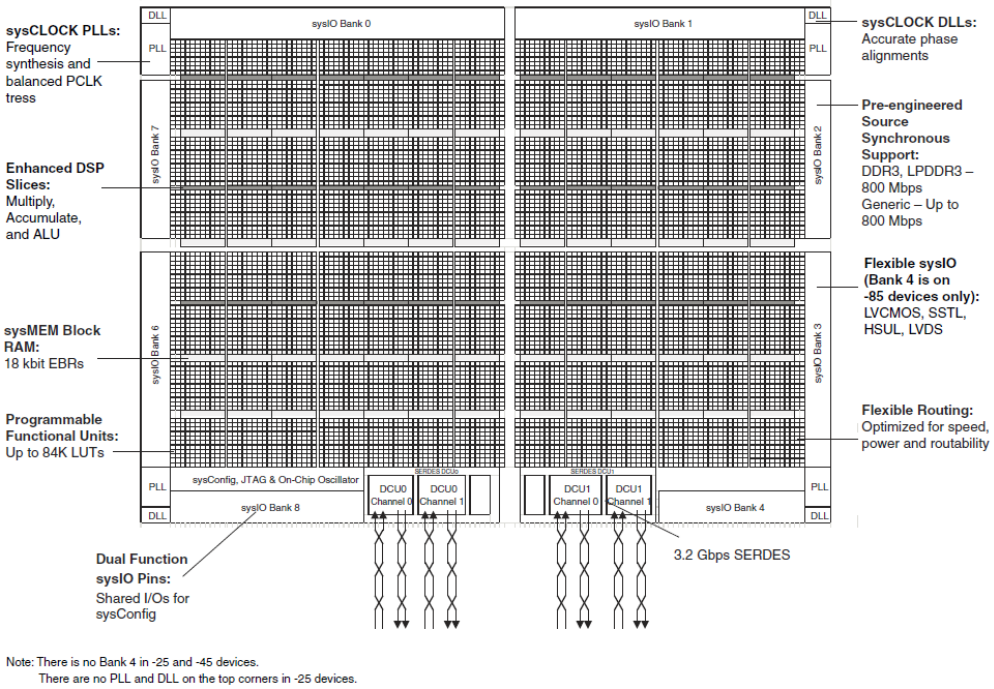


Figure 9.1: Evaluation board of the LFE5UM5G-85F FPGA: a variant of the ECP5.[1]



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 9.2: Architecture of the ECP5 FPGA. Note that it is tile-based in a square grid structure.[2]

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 compass direction, a 2-bit LUT and an 1-bit register. It can be configured in several ways:

1. As a single wire from one compass direction to the opposite
2. As a cross-section of two orthogonal wires
3. A wire from direction A making a 90 degree turn to direction B, and a wire from direction B to the opposite direction of A.
4. As a configurable 2-input-2-output LUT with any two compass directions as inputs and the other two as outputs
5. As an 1-bit register with any compass direction as data input, any other compass direction as clock-enable (allowing the register value to be set) and any single remaining compass direction as register output. The final compass direction outputs the input of the register.

These different configurations are visualised in Figure 9.3.

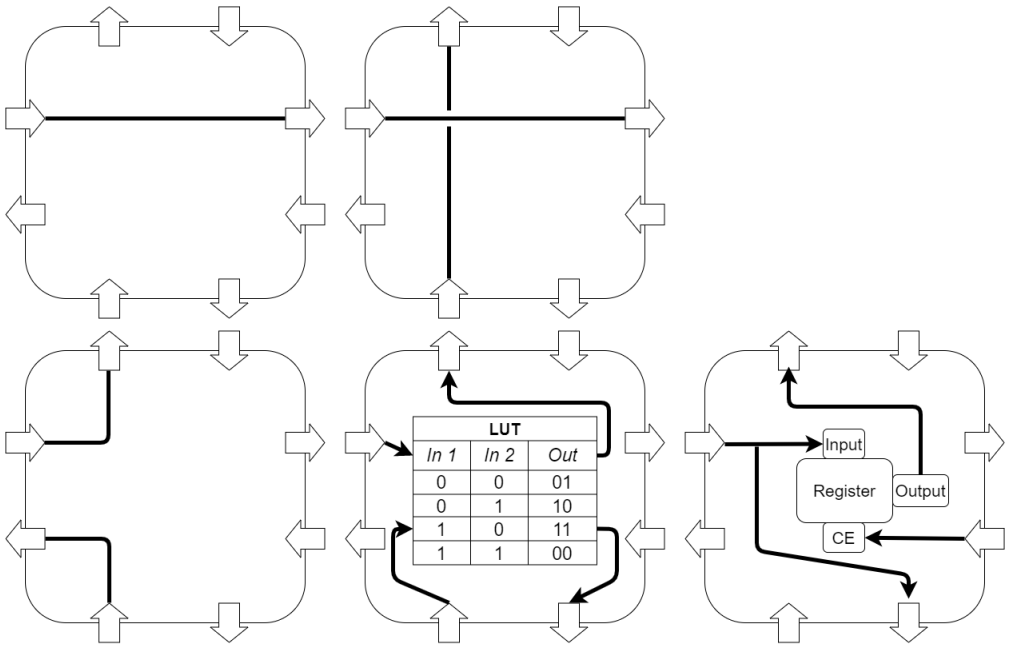


Figure 9.3: All five possible configurations of VirBoard.

There are countless ways to model this functionality using wires, transistors and logic cells, each of which would yield working emulations if a subgraph homeomorphism of the model is found in the graph model of the concrete FPGA. In principle, one could enumerate all such models and attempt to find subgraph homeomorphisms with each of them, maximizing the expected result. For the scope of our research, however, we limit ourselves to a single graph model that performs well for subgraph homeomorphism search. For this purpose, we design the model of the virtual FPGA with two goals in mind:

1. It contains as few vertices and edges as possible, so that the search space is as small as possible.
2. It contains as few rare resources as possible, so that more subgraph homeomorphisms are present.

We designed a graph model of VirBoard that encompasses all listed functionality and contains a total of 30 vertices and 32 edges. We were not able to produce a smaller model or one that uses fewer resources. This graph model of VirBoard is shown in Figure 9.4.

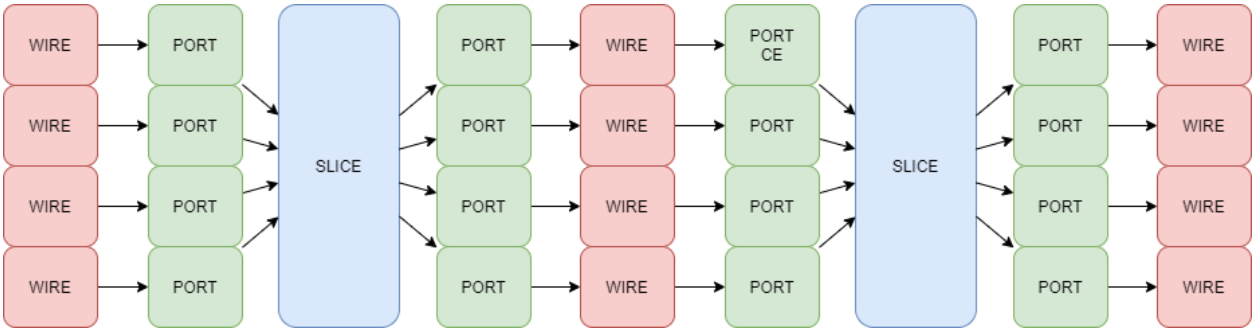


Figure 9.4: A graph model of a single cell of the virtual FPGA aimed to emulate on the Lattice ECP5

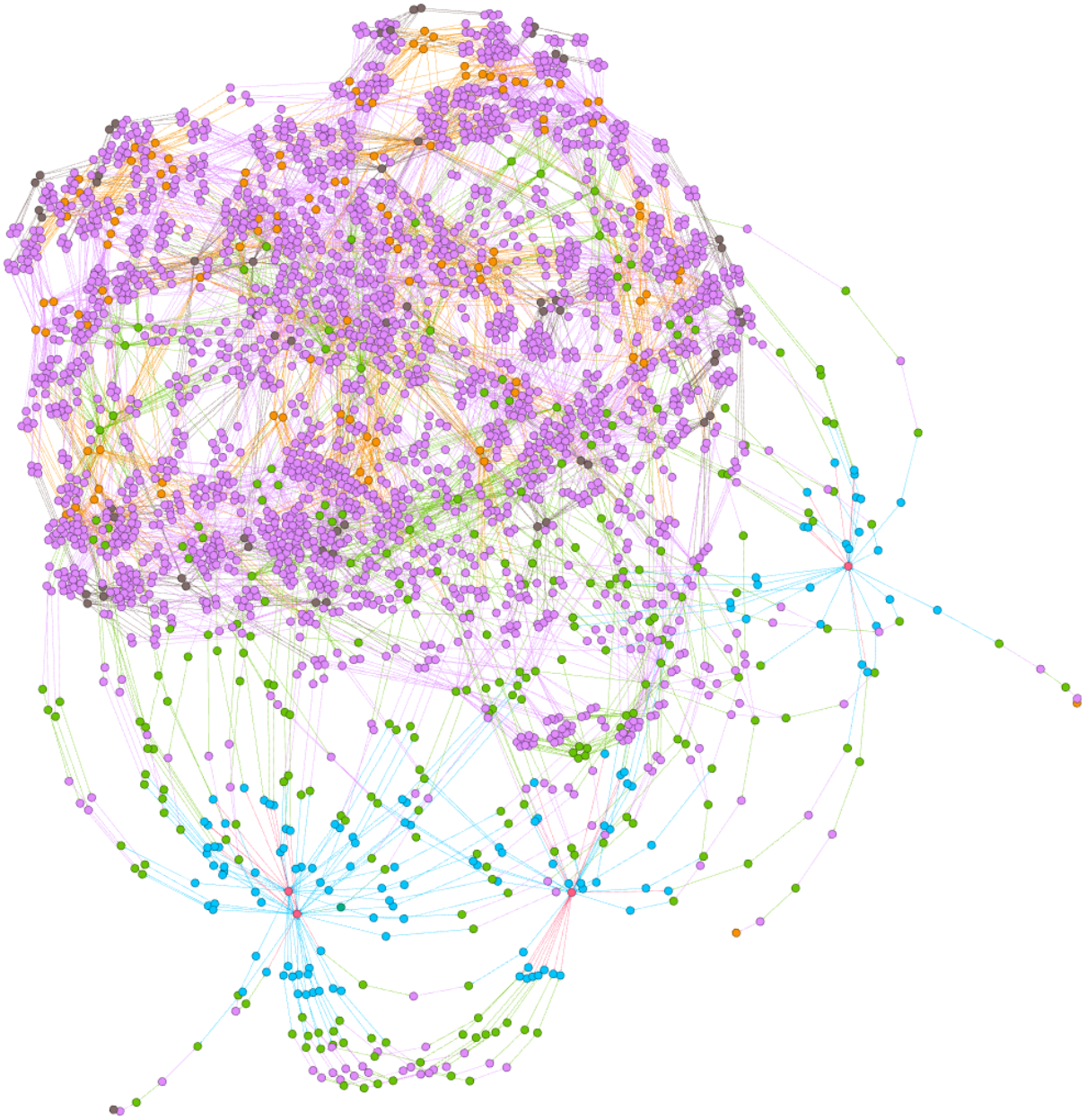


Figure 9.5: Graph model of a single ECP5 logic tile (out of ±42000 logic tiles). Transistors are purple, logical units are red, ports are blue, wires are orange (except those that are neighbouring an adjacent cell, which are brown).

10 EXPERIMENTS

Using a machine with an AMD Ryzen 5 3600 CPU @ 3.7GHz and 32GB 3200MHz memory[1] we conduct several experiments with different inputs and settings. For this purpose, we generate random pairs of source graph- and target graph that have subgraph homeomorphisms in them and random pairs that do not.

Firstly, we pose a method for generating random graphs representing virtual FPGAs that adhere to our FPGA modeling methodology:

1. From $|V|$ we establish how many wires, transistors, logic cells and ports it needs to have the same distribution as the virtual FPGA.
2. We apply the appropriate labels to the vertex set as specified in Chapter 5.
3. We connect each port to a random logic cell in a random direction (except the `CE` port which has to be an input). If the graph is so small there is no logic cell, we add a single one.
4. We connect each port to a random wire in the appropriate direction.
5. We connect each transistor vertex to two different random wire vertices (one with an incoming edge and the other with an outgoing edge).

Since vertex disjoint subgraph homeomorphisms are rare in random graph pairs, we artificially construct source graph-target graph pairs such that it is guaranteed a subgraph homeomorphism exists:

1. First, we generate a random source graph with V_S vertices using the 5-step method described above.
2. Then, we establish a distribution of $|V_T|$ wires, transistors, logic cells and ports we need to add to resemble the FPGA use case target graph distribution as closely as possible, and add missing vertices to the source graph such that the total vertex set encompasses the established distribution.
3. We apply the appropriate labels to these extra vertices as specified in Chapter 5.

[1] Comparitive experiments were sometimes run on a machine with different specifications

4. We connect each new port vertex to a random logic cell vertex, prioritizing new logic cell vertices over existing ones while the new logic cell vertices have a lower average degree. Furthermore, we connect each new port vertex to a random wire in the appriopriate direction.
5. We use half of the extra wires to intersect existing connections. With each intersection equally likely, we intersect a $(\texttt{WIRE} \to \texttt{ARC} \to \texttt{WIRE})$-connection into a $(\texttt{WIRE} \to \texttt{ARC} \to \texttt{WIRE} \to \texttt{ARC} \to \texttt{WIRE})$-connection, intersect a $(\texttt{PORT} \to \texttt{WIRE})$-connection into a $(\texttt{PORT} \to \texttt{WIRE} \to \texttt{ARC} \to \texttt{WIRE})$-connection or intersect a $(\texttt{WIRE} \to \texttt{PORT})$-connection into a $(\texttt{WIRE} \to \texttt{ARC} \to \texttt{WIRE} \to \texttt{PORT})$-connection.
6. We connect each remaining transistor vertex to two different random wire vertices (one with an incoming edge and the other with an outgoing edge), connecting with at least one new wire vertex while the new wire vertices have a lower average degree.

Using this method we obtain random test cases that resemble the graphs of the FPGA use case as closely as possible while still introducing some randomness that allow us to benchmark our algorithm's performance under a variety of optimisations.

Firstly, we compare the different methods of path iteration. The time taken to find a homeomorphism gives an indication of the overhead introduced by a path iterator and also taking the heuristic value of that path iterator into account. This comparison is shown in Figure 10.1. We compare the K-path, control point (CP), DFS, Greedy cached DFS (GDFS C), greedy in-place DFS using a distance metric (GDFS O IP) and greedy in-place DFS using a partial-mapping-aware distance metric (GDFS A IP) path iteration methods. We used target graphs that were respectively 50%, 200% and 400% larger than the souce graph[2]. We also add the performance of a portfolio method: running the algorithm in parallel with each path iteration method and choosing the fastest outcome for each individual test case. We measure the space usage of each path iterator, the results of which are shown in Figure 10.2.

Without optimisations, we observe exponential behaviour with approximate complexity$O(e^{0.7*|V_S|})$ for $|V_T| = 1\frac{1}{2}|V_S|$, $O(e^{1.0482*|V_S|})$ for $|V_T| = 3|V_S|$ and $O(e^{1.133*|V_S|})$ for $|V_T| = 5|V_S|$. Extrapolating this to the FPGA use case where $|V_T| = 97|V_S|$ assuming a logarithmic trend, we get a scalability of $\pm O(e^{2.57442*|V_S|})$ for our FPGA use case without pruning or contraction.

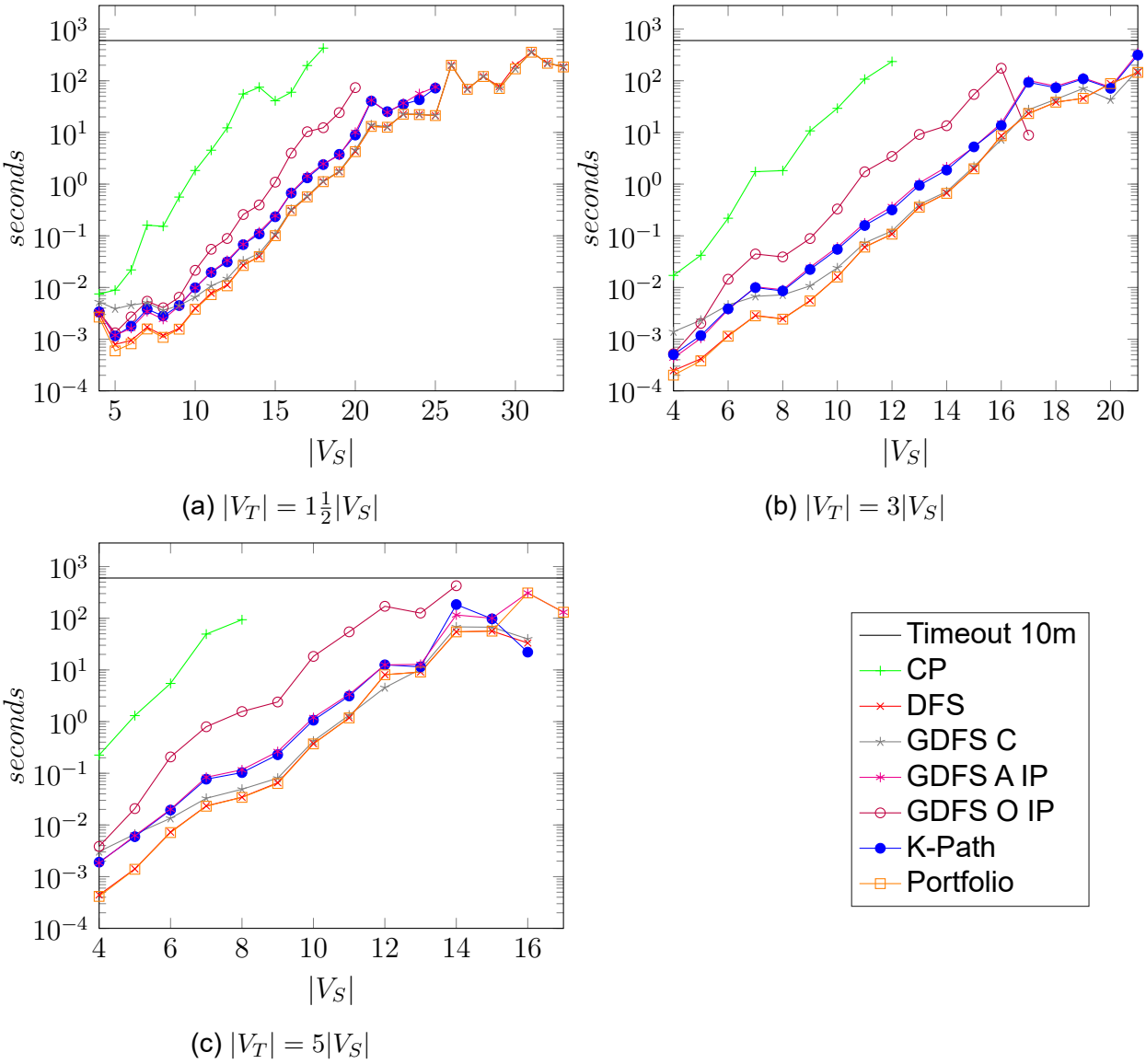
We measure the performance difference between using GreatestConstrainedFirst as the source graph vertex ordering compared to a random ordering in Figure 10.3. Similarly, we compare the distance-based target graph vertex ordering with a degree-based ordering in Figure 10.4 and compare the degree-based ordering with a random ordering in Figure 10.5.

Next, we measure the performance gain from contraction by plotting the mean time increase or decrease in cases where subgraph homeomorphisms are present in Figure

[2]In the FPGA use case the target graph is $\pm$ 97 times larger than the souce graph. This, however, is too computationally expensive for our testing methodology.

10.6. Similarly, we measure the performance gain from each method of pruning (pruning technique, filtering and application) compared to applying no pruning at all in Figures 10.7 through 10.14 and measure its impact on space usage in Figures 10.15 and 10.16.

Lastly, we take the optimal set of settings and measure method how long the algorithm on average takes to find a subgraph homeomorphism where one exists, the results of which are shown in Figure 10.17. The results for $|V_T| = 97|V_S|$ can be approximated with the exponential formula $t = 0.0064e^{0.6204*|V_S|}$; this gives us an estimation for the business case with $|V_S| = 30$ of $\pm$ 6 days of computation time.



(a) $|V_T| = 1\frac{1}{2}|V_S|$

(b) $|V_T| = 3|V_S|$

(c) $|V_T| = 5|V_S|$

Figure 10.1: Performance of using different path iterators in test cases with present subgraph homeomorphisms. We avoid unnecessarily long paths, and use no pruning or contraction. We include a portfolio method that takes the best performance rating for each individual test case.

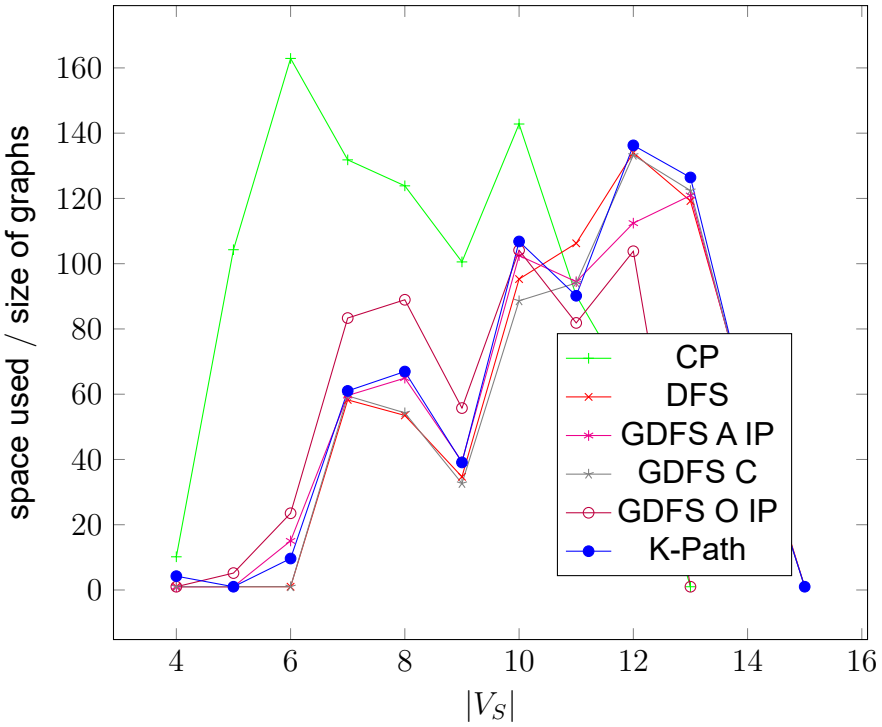


Figure 10.2: Space usage of the algorithm with each path iterator relative to the space usage of the input graphs. Unnecessarily long paths are avoided, target vertices are ordered by degree and contraction is disabled.

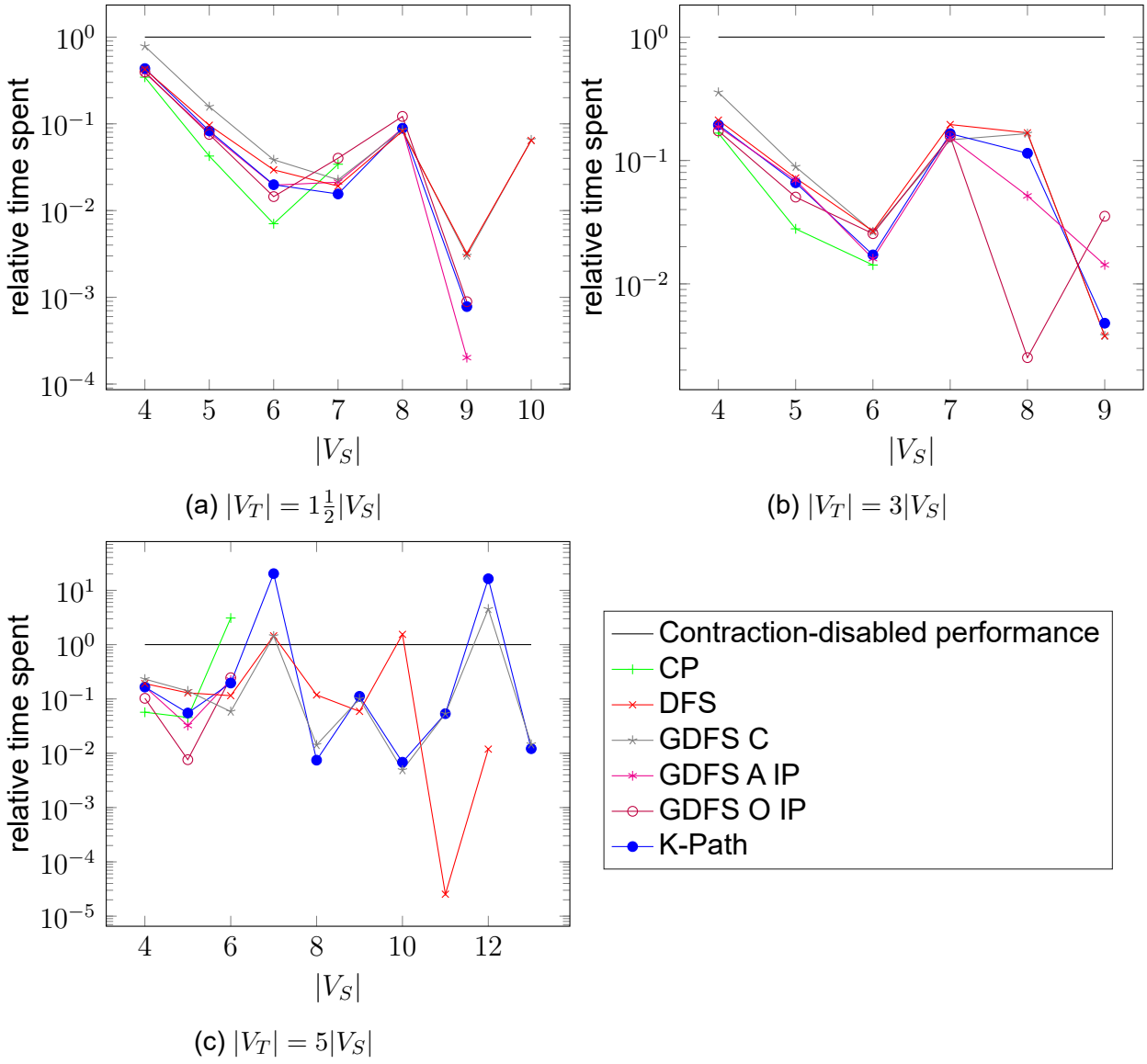


Figure 10.3: Performance of our algorithm with the GreatestConstrainedFirst source graph vertex order relative to the performance of the algorithm with a random source graph vertex order. We avoid unnecessarily long paths, do not perform contraction and use no pruning. Data points above the black reference line denote the GreatestConstrainedFirst ordering introduces more delay, and data points below the reference line denote that it saves time. Note the logarithmic y-axis.

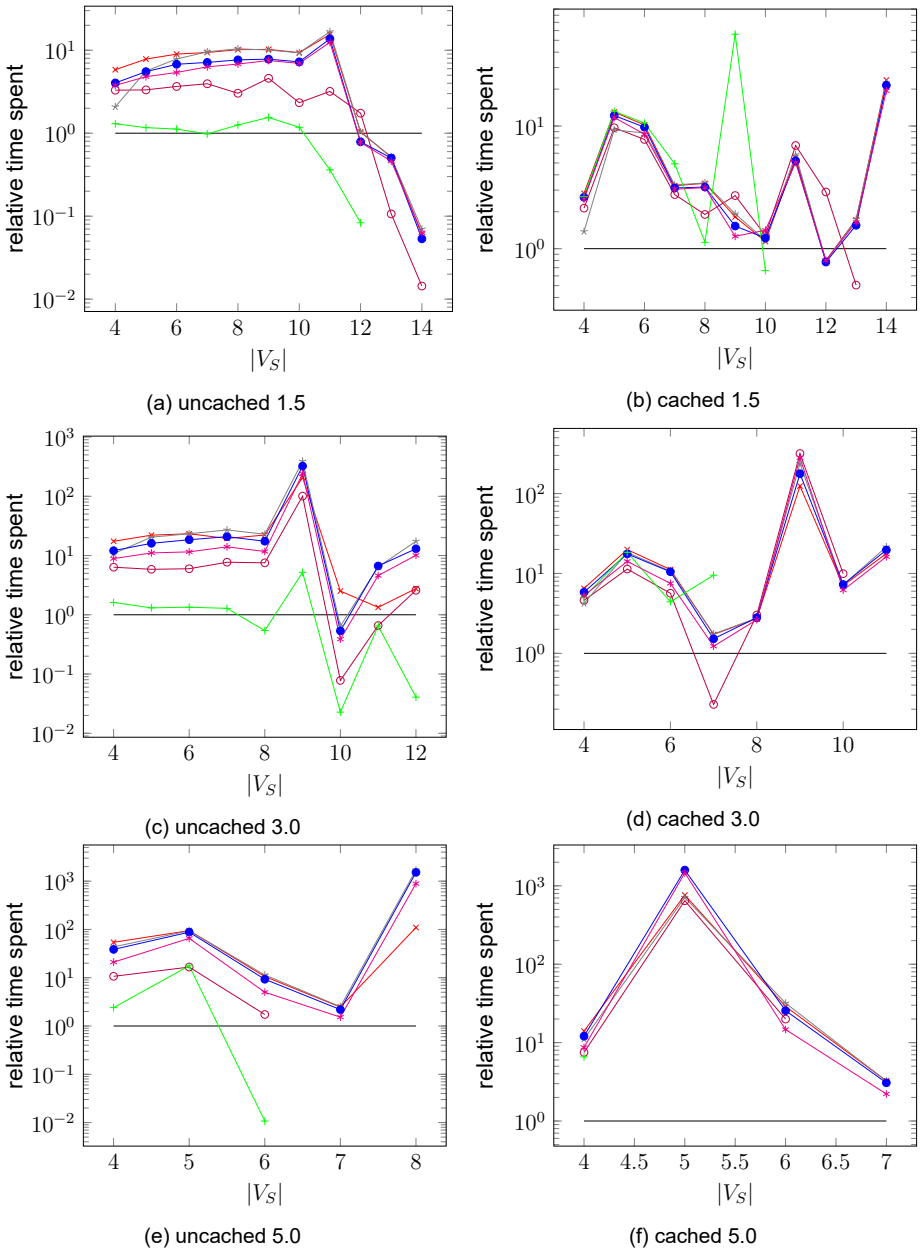
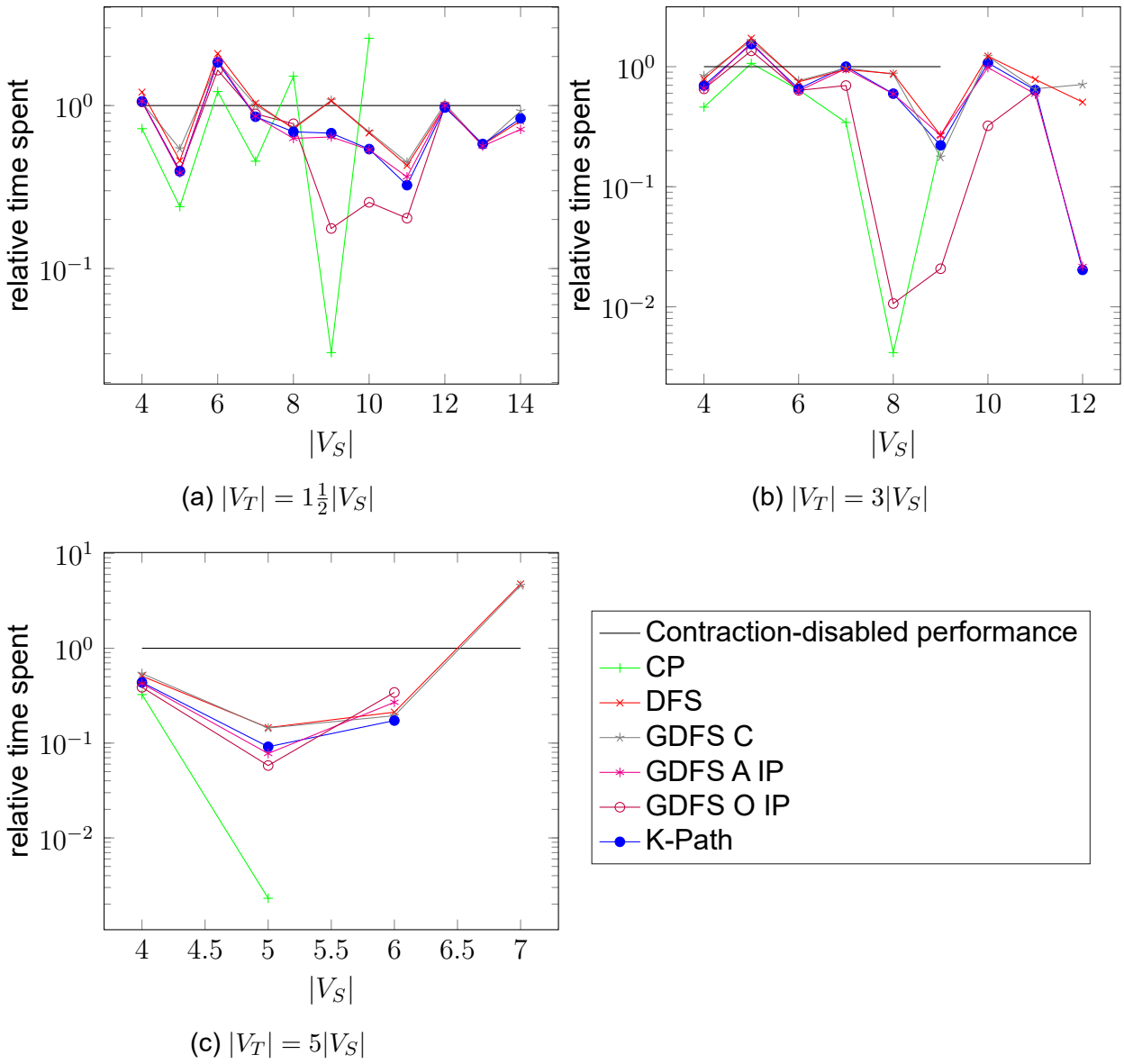


Figure 10.4: Performance of our algorithm with the distance based target graph vertex order relative to the performance of the algorithm with the degree-based target graph vertex order. We avoid unnecessarily long paths, do not perform contraction and use no pruning. Data points above the black reference line denote that this ordering introduces more delay, and data points below the reference line denote that this ordering saves time. Note the logarithmic y-axis.

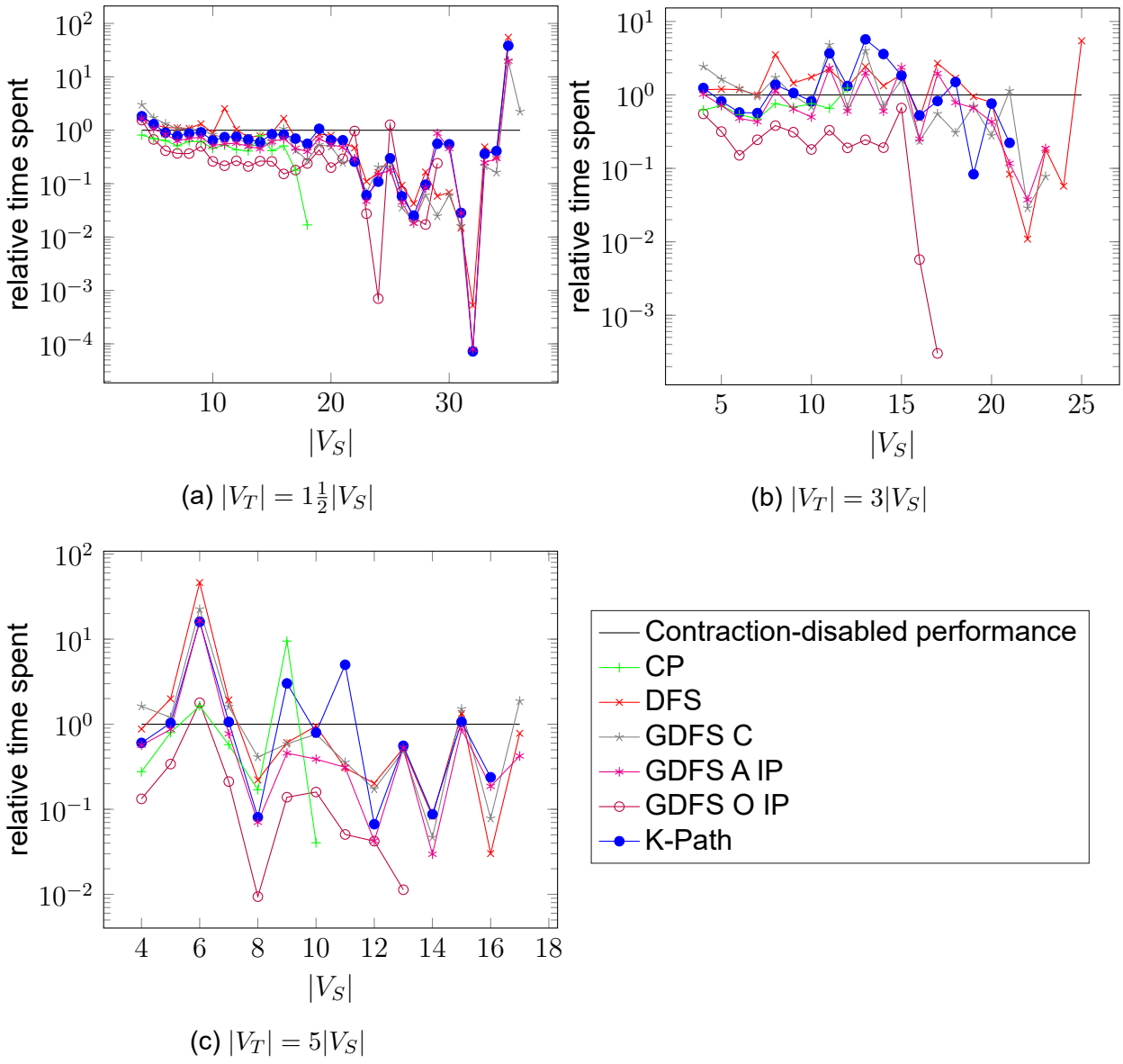


(a) $|V_T| = 1\frac{1}{2}|V_S|$

(b) $|V_T| = 3|V_S|$

(c) $|V_T| = 5|V_S|$

Figure 10.5: Performance of our algorithm with the degree-based target graph vertex order relative to the performance of the algorithm with a random target graph vertex order. We avoid unnecessarily long paths, do not perform contraction and use no pruning. Data points above the black reference line denote the degree-based ordering introduces more delay, and data points below the reference line denote that it saves time. Note the logarithmic y-axis.



(a) $|V_T| = 1\frac{1}{2}|V_S|$

(b) $|V_T| = 3|V_S|$

(c) $|V_T| = 5|V_S|$

Figure 10.6: Mean relative time consumption of contraction-enabled subgraph homeomorphism search compared to contraction-disabled for different path iteration methods. For data points below the reference line contraction saves time while for data points above it contraction costs extra time. We handled a maximum of 1000 test cases or 30 minutes per value of $|V_S|$ for each path iteration method.

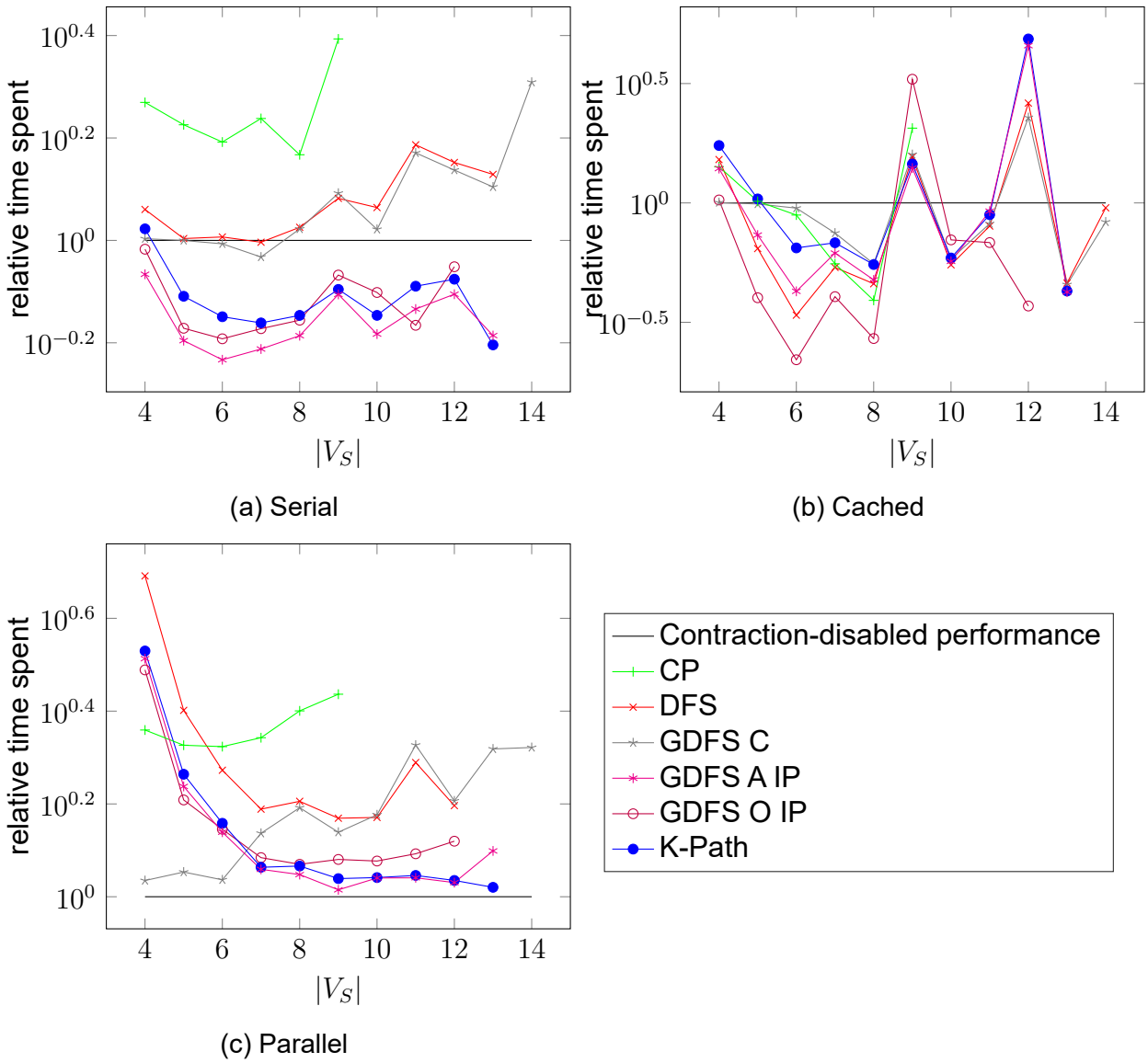


Figure 10.7: Performance of our algorithm with **ZeroDomain** pruning using **'Labels and neighbours'** filtering relative to the performance of the algorithm without pruning. We avoid unnecessarily long paths, do not perform contraction and use the degree-based target graph vertex ordering. Data points above the black reference line denote that the pruning method introduces more delay, and data points below the reference line denote that the pruning method saves time. Note the logarithmic y-axis.

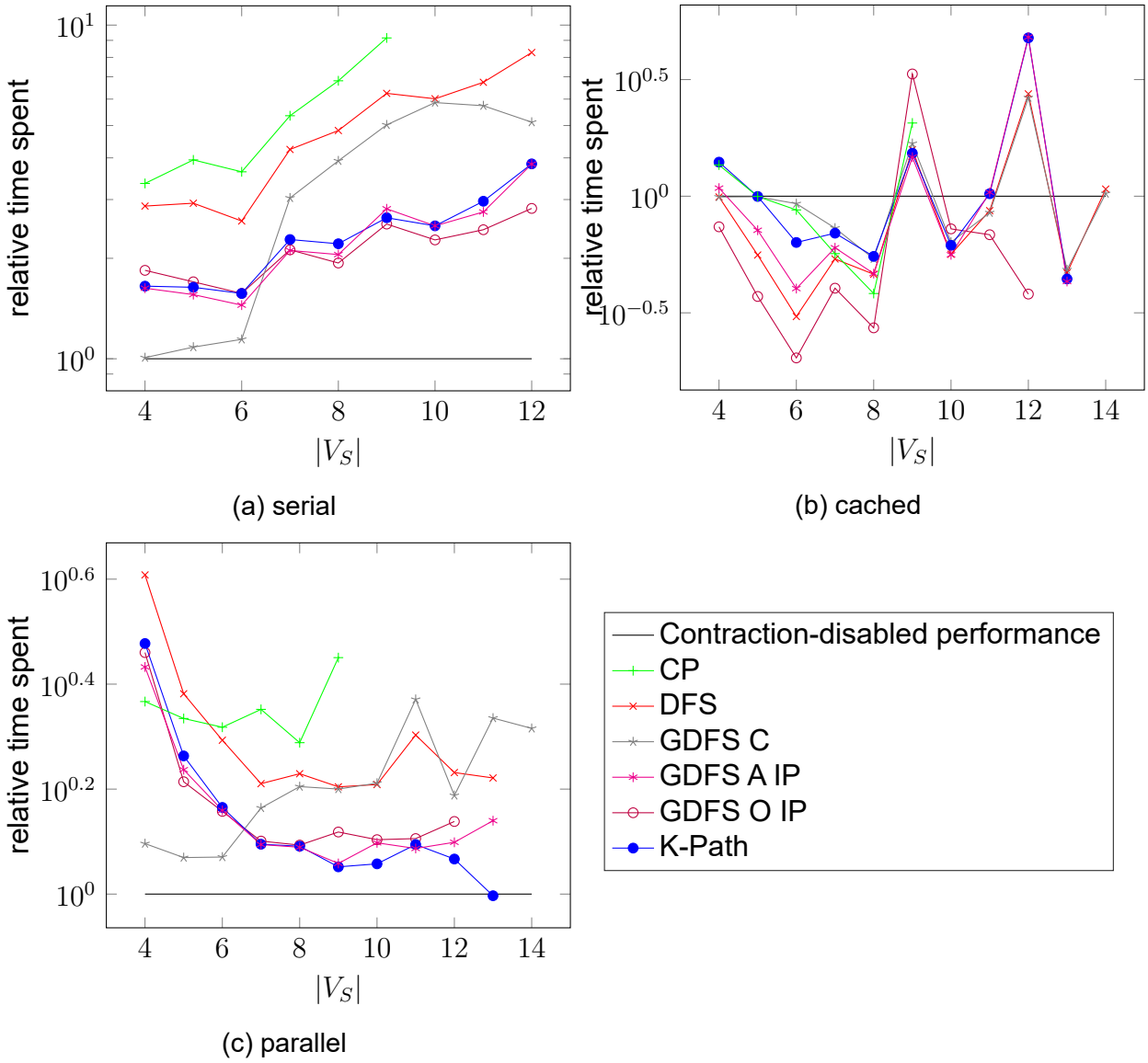


Figure 10.8: Performance of our algorithm with **ZeroDomain** pruning using **'Free neighbours'** filtering relative to the performance of the algorithm without pruning. We avoid unnecessarily long paths, do not perform contraction and use the degree-based target graph vertex ordering. Data points above the black reference line denote that the pruning method introduces more delay, and data points below the reference line denote that the pruning method saves time. Note the logarithmic y-axis.

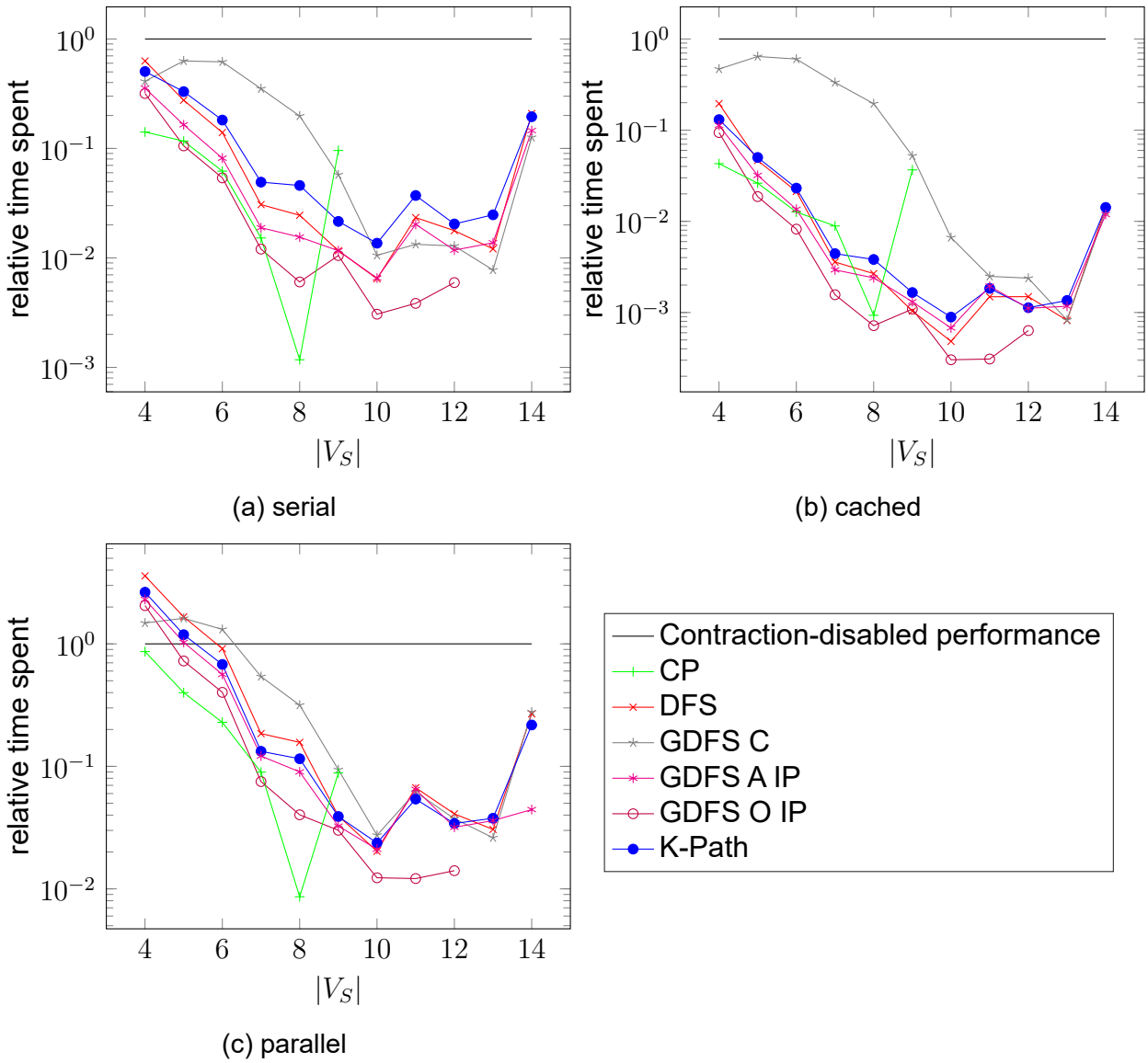


Figure 10.9: Performance of our algorithm with **ZeroDomain** pruning using **'M-filtering'** filtering relative to the performance of the algorithm without pruning. We avoid unnecessarily long paths, do not perform contraction and use the degree-based target graph vertex ordering. Data points above the black reference line denote that the pruning method introduces more delay, and data points below the reference line denote that the pruning method saves time. Note the logarithmic y-axis.

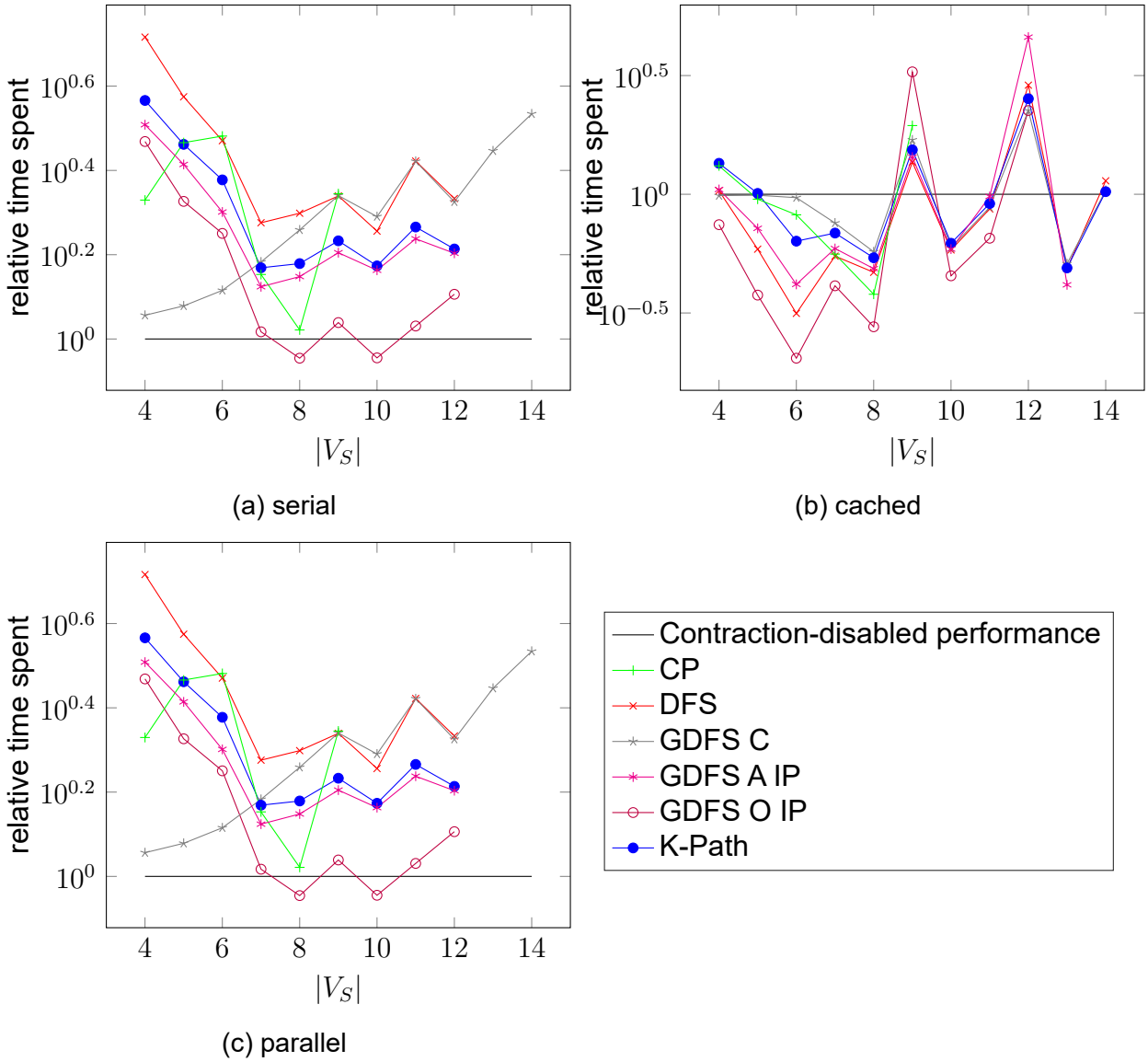


Figure 10.10: Performance of our algorithm with **ZeroDomain** pruning using **'N-Filtering'** filtering relative to the performance of the algorithm without pruning. We avoid unnecessarily long paths, do not perform contraction and use the degree-based target graph vertex ordering. Data points above the black reference line denote that the pruning method introduces more delay, and data points below the reference line denote that the pruning method saves time. Note the logarithmic y-axis.

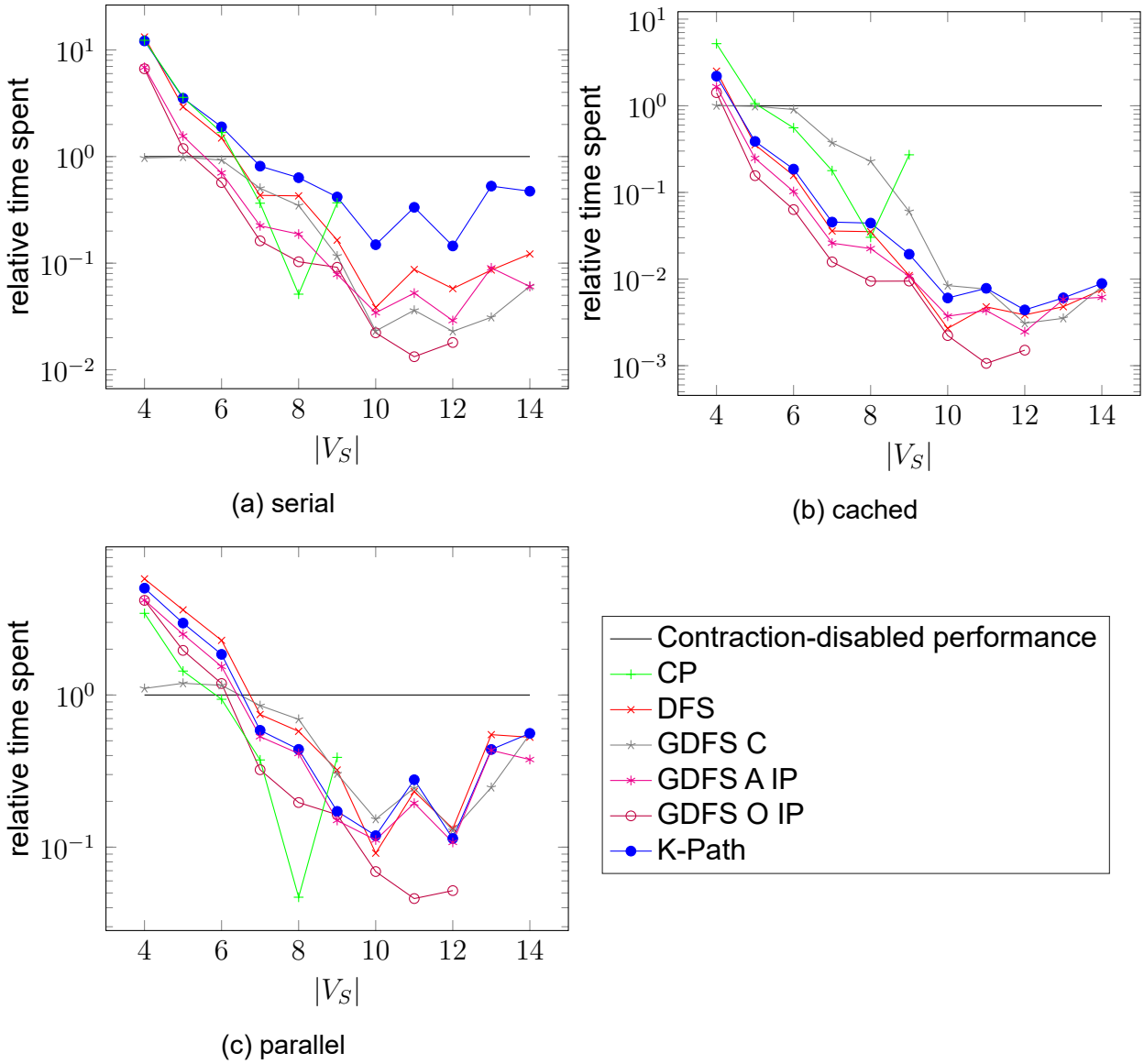


Figure 10.11: Performance of our algorithm with **AllDifferent** pruning using **'Labels and neighbours'** filtering relative to the performance of the algorithm without pruning. We avoid unnecessarily long paths, do not perform contraction and use the degree-based target graph vertex ordering. Data points above the black reference line denote that the pruning method introduces more delay, and data points below the reference line denote that the pruning method saves time. Note the logarithmic y-axis.

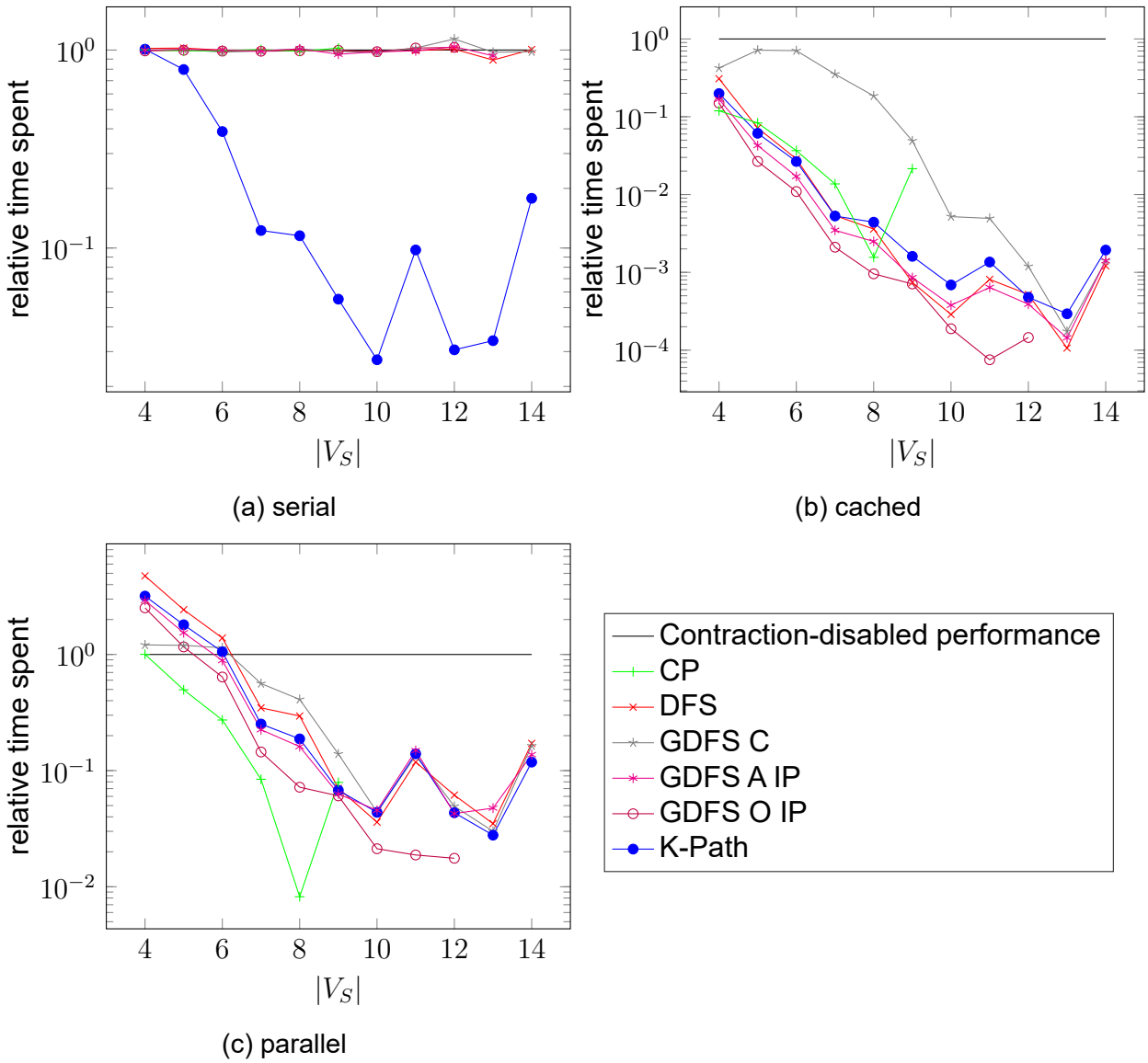


Figure 10.12: Performance of our algorithm with **AllDifferent** pruning using **'Free neighbours'** filtering relative to the performance of the algorithm without pruning. We avoid unnecessarily long paths, do not perform contraction and use the degree-based target graph vertex ordering. Data points above the black reference line denote that the pruning method introduces more delay, and data points below the reference line denote that the pruning method saves time. Note the logarithmic y-axis.

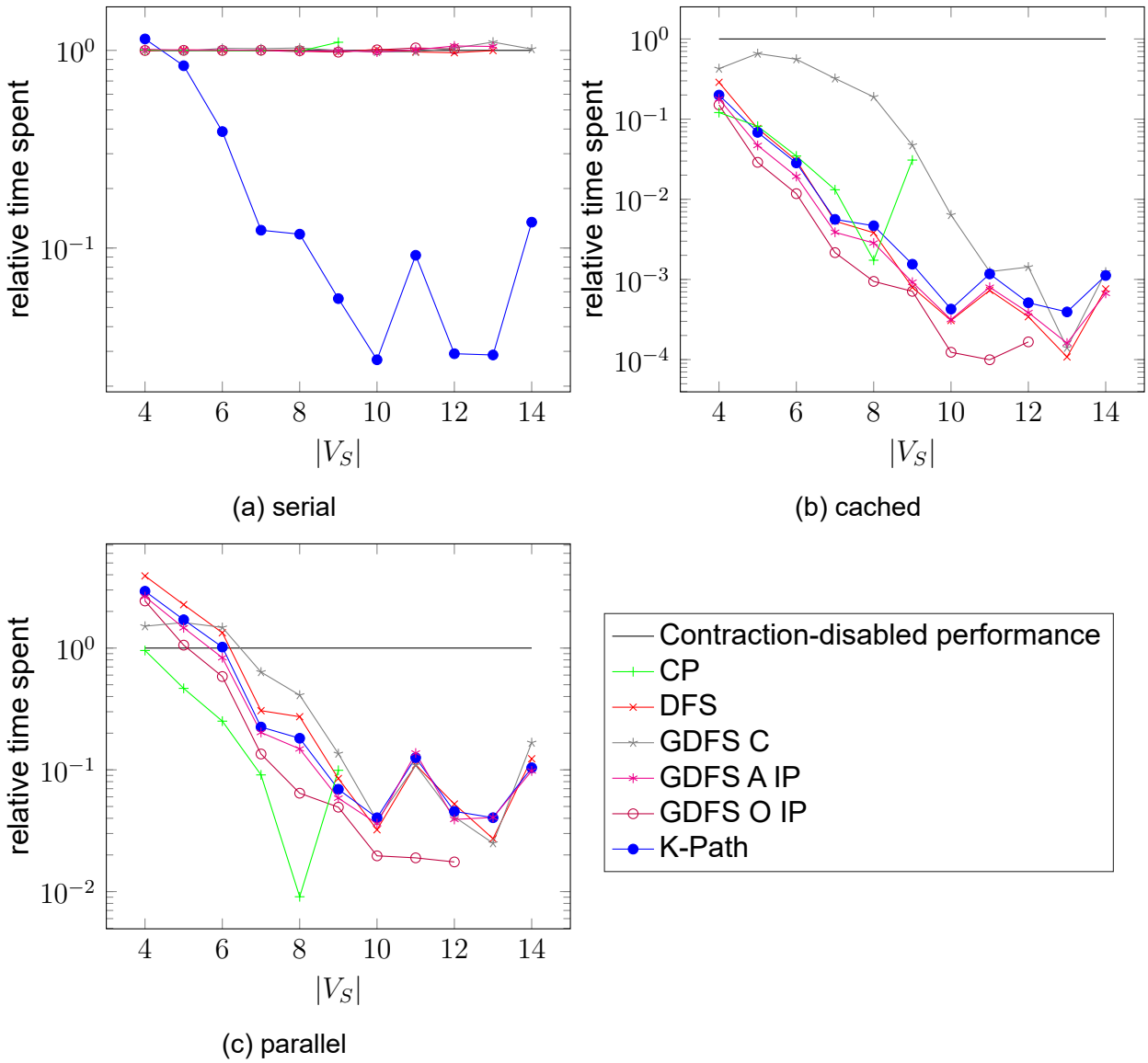


Figure 10.13: Performance of our algorithm with **AllDifferent** pruning using **'M-filtering'** filtering relative to the performance of the algorithm without pruning. We avoid unnecessarily long paths, do not perform contraction and use the degree-based target graph vertex ordering. Data points above the black reference line denote that the pruning method introduces more delay, and data points below the reference line denote that the pruning method saves time. Note the logarithmic y-axis.

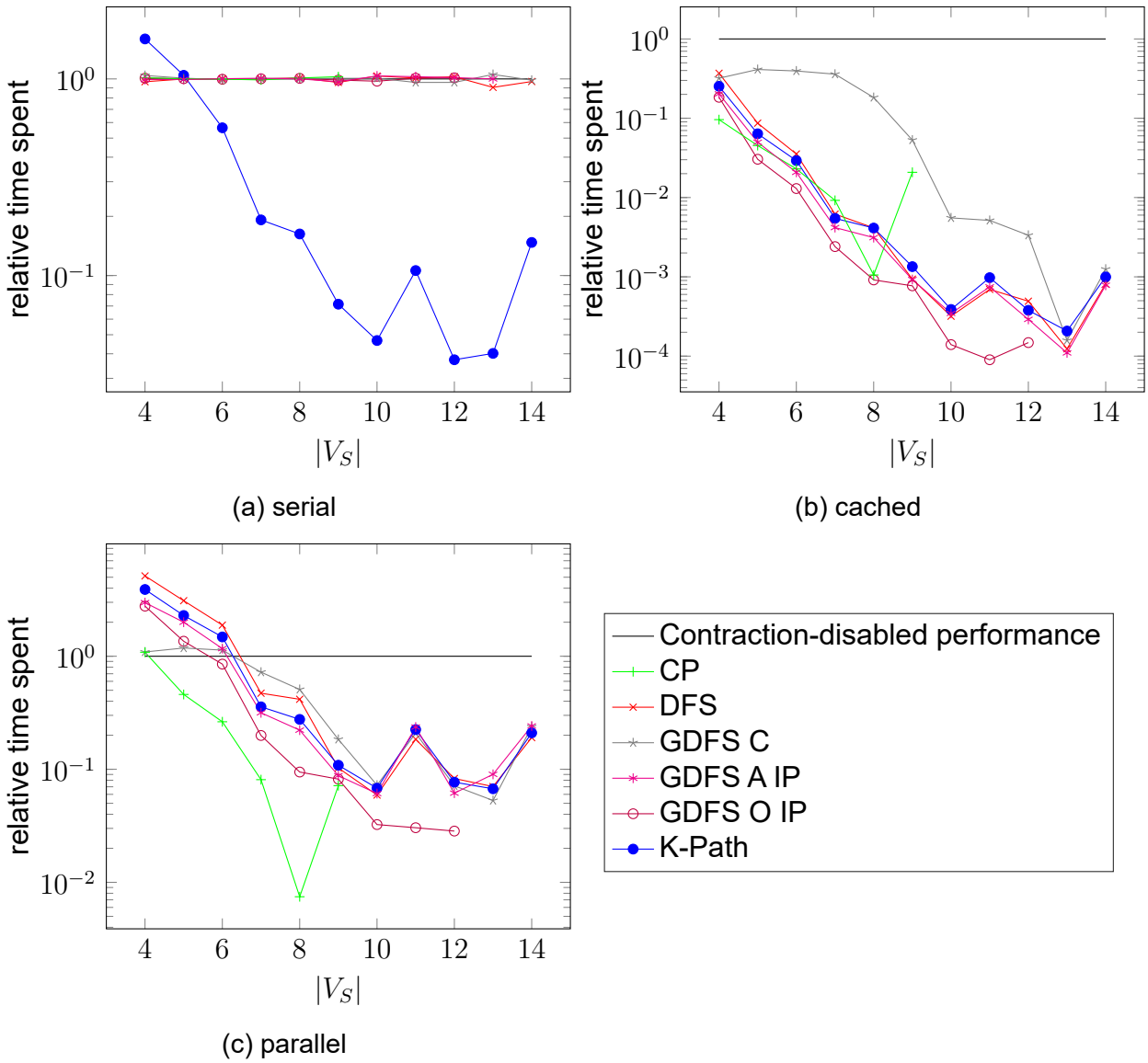


Figure 10.14: Performance of our algorithm with **AllDifferent** pruning using **'N-filtering'** filtering relative to the performance of the algorithm without pruning. We avoid unnecessarily long paths, do not perform contraction and use the degree-based target graph vertex ordering. Data points above the black reference line denote that the pruning method introduces more delay, and data points below the reference line denote that the pruning method saves time. Note the logarithmic y-axis.

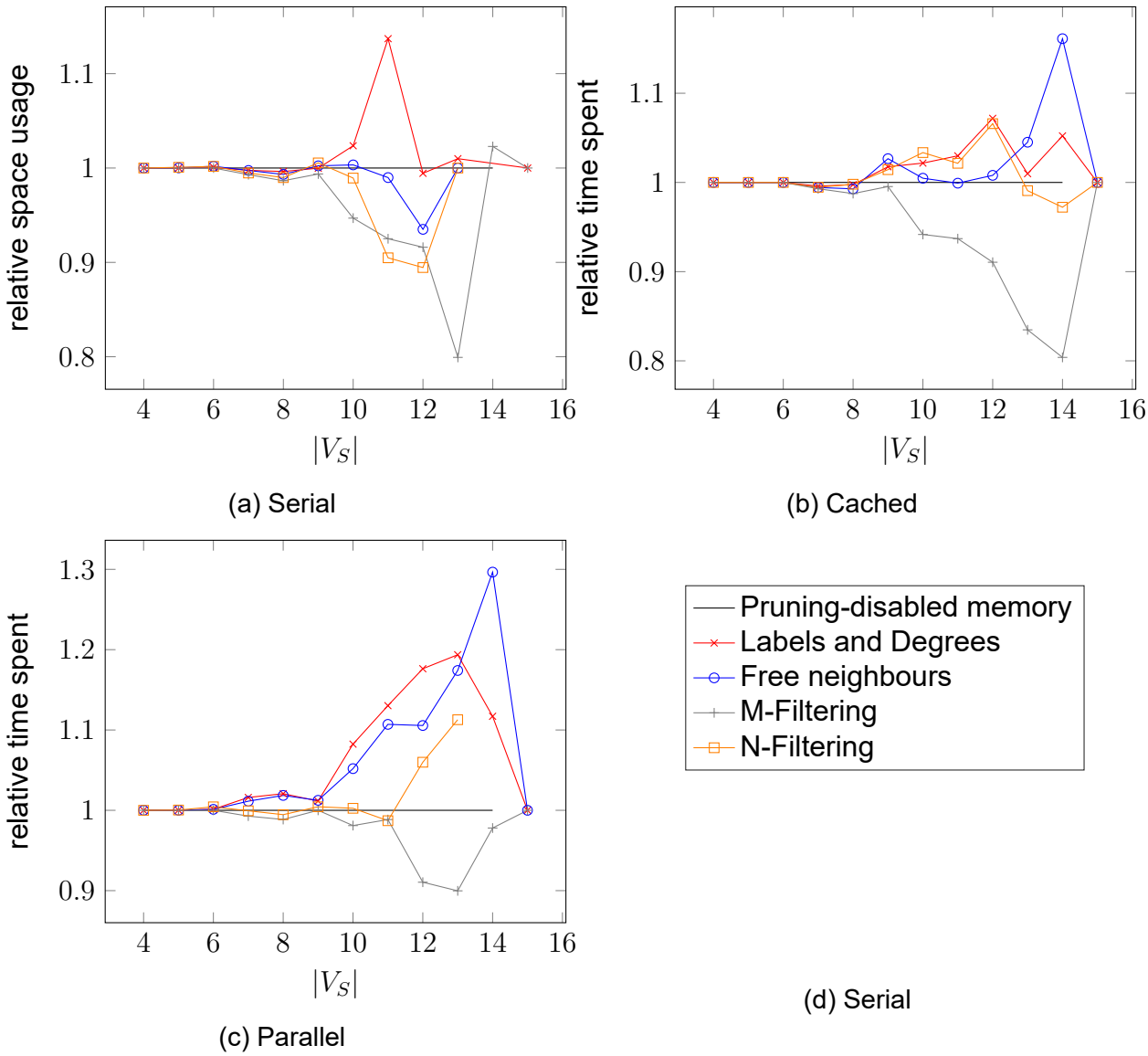


Figure 10.15: Memory usage of our algorithm with **ZeroDomain** pruning using various filtering methods compared to memory usage without pruning. We avoid unnecessarily long paths, do not perform contraction, use the degree-based target vertex ordering and use DFS path iteration. Data points above the black reference line denote that the pruning method increases memory usage and data points below the reference line denote that the pruning method saves memory.

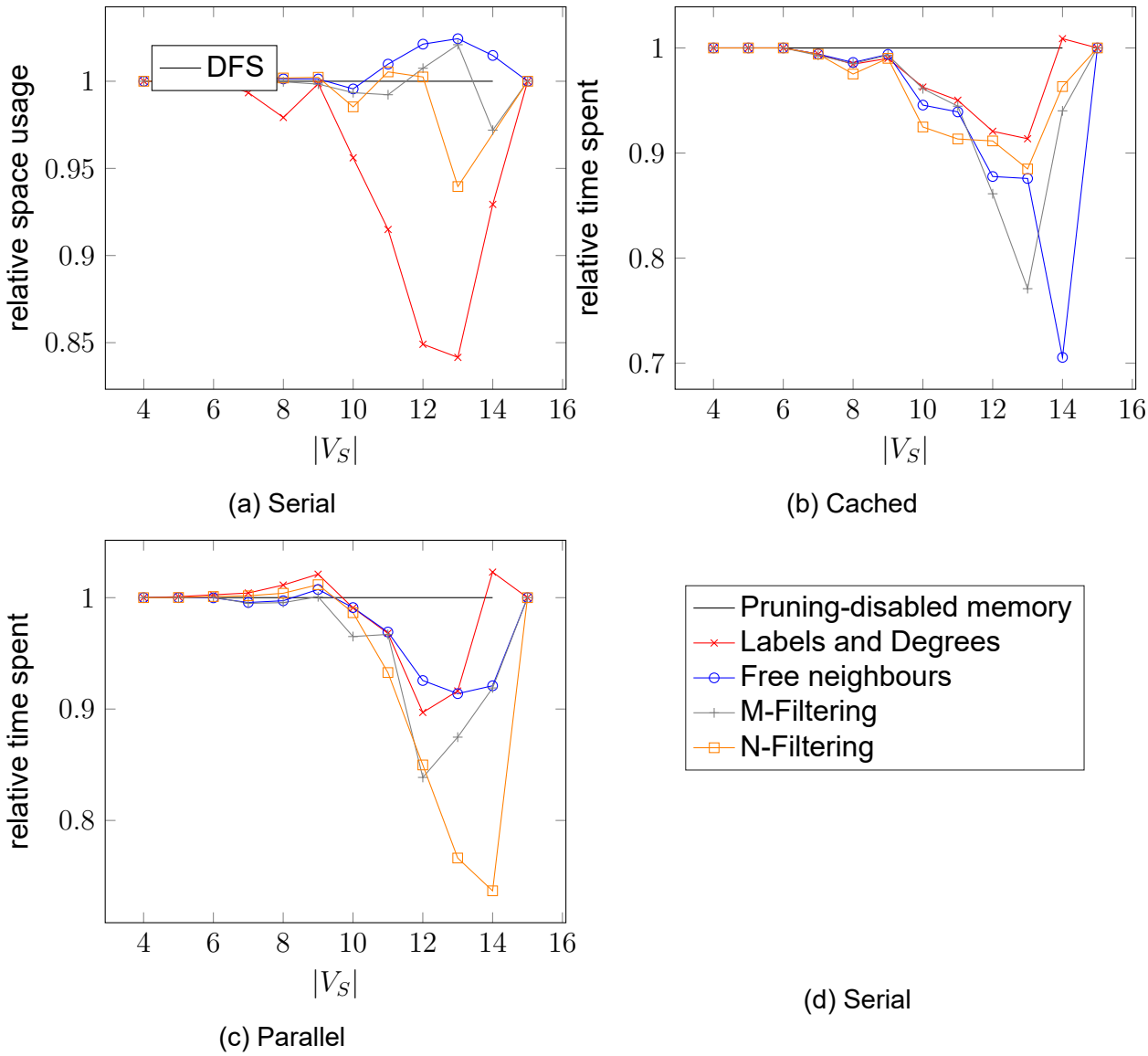


Figure 10.16: Memory usage of our algorithm with **AllDifferent** pruning using various filtering methods compared to memory usage without pruning. We avoid unnecessarily long paths, do not perform contraction, use the degree-based target vertex ordering and use DFS path iteration. Data points above the black reference line denote that the pruning method increases memory usage and data points below the reference line denote that the pruning method saves memory.

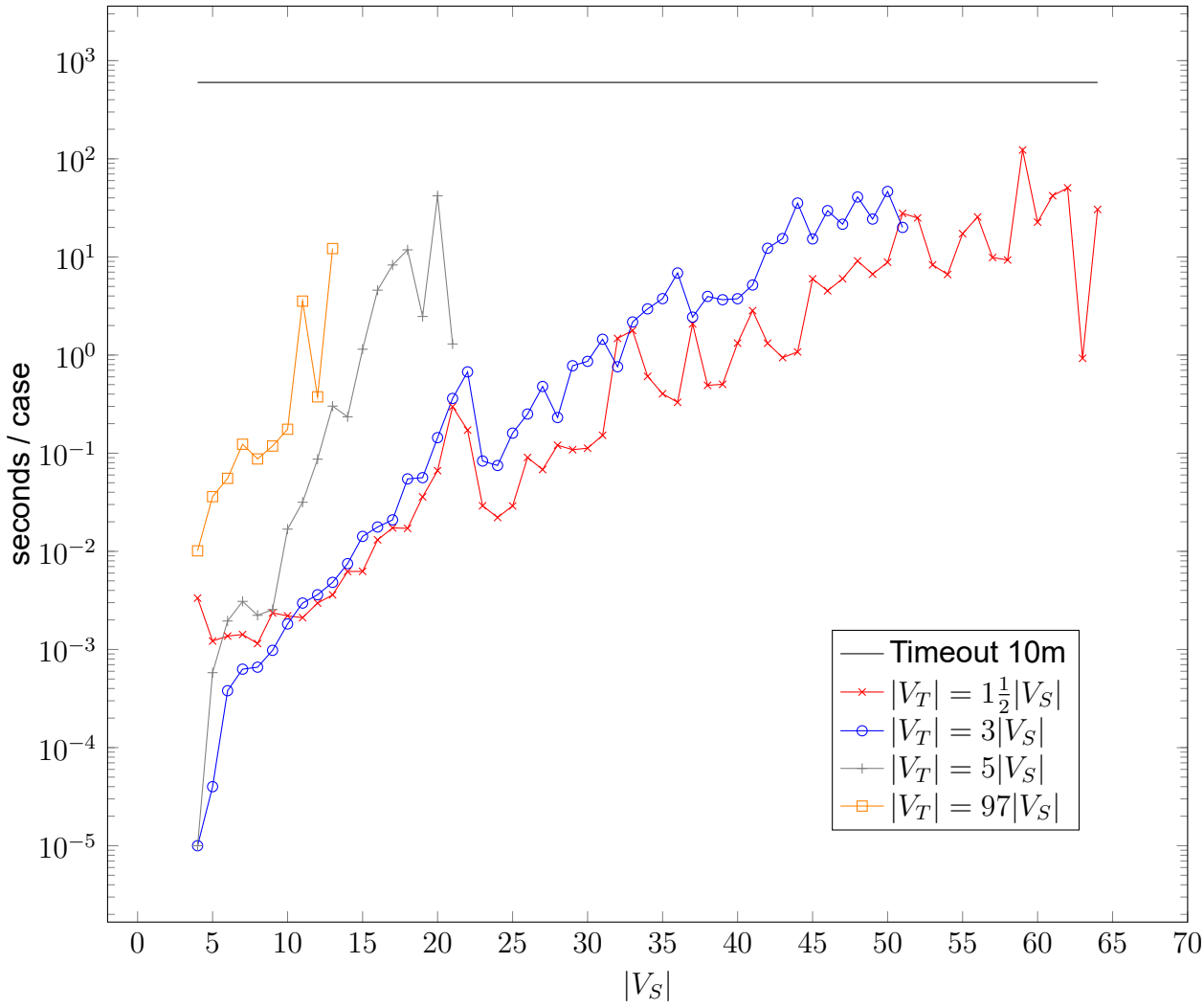


Figure 10.17: Performance of our algorithm *ndsh2 – contract* using the configurations that individually perform best: DFS path iteration, avoiding unnecessarily long paths and N-reachability AllDifferent pruning. Two instances are run in parallel, one of which uses contraction and one of which does not: the fastest of the two is used. For use 10 minutes worth of tests for each data point and stop testing when that is not enough for finding a homeomorphism in a single test case.

11 DISCUSSION

From our experiments, we found that in-place depth first search path iteration is (of the methods compared) the most efficient path iteration method to replace Xiao's structure with precomputed paths for our business case. We could not find enough data to make conclusions on how memory usage scales with each path iteration method; however, all path iteration methods (except for control point iteration) seem to scale approximately equally within our graph size range.

Our experiments show that the GreatestConstrainedFirst source graph vertex ordering from RI-DS [8] is indeed more efficient than the assumed random ordering of Xiao's algorithm. Moreover, the experiments show that ordering target vertices by degree introduced by Glasgow [41] is on average more efficient than the assumed random ordering of Xiao's algorithm or the distance-based ordering we introduced.

We compared each of our 24 pruning methods and conclude that cached AllDifferent pruning with N-filtering is the most computationally effective pruning method for our business case. This is partly in line with Xiao's algorithm, which uses cached ZeroDomain pruning with N-filtering.

With regards to space usage, we observe that the fastest pruning method (i.e. cached AllDifferent pruning with N-filtering) saves space, even though it requires caching the domains obtained through N-filtering. This could be the result of the pruner pruning search space branches in which many resources are being used and would require much memory.

For applications of our algorithm outside of FPGA emulation, different settings may yield better performance. Our test cases are specifically designed to represent graphs that adhere to our FPGA graph model, and may have properties that graphs from a different domain do not have. Therefore, verifying the appropriate settings for a different business case is adviseable if computational requirements are relevant.

The performance of our set of optimal configurations performs well enough for our FPGA business case. From the test results we estimated to require 6 days of computation for our business case or cases of comparable size. Computing this once for each pair of virtual FPGA and concrete FPGA is reasonable.

A disadvantage of our approach is that it requires modelling the virtual FPGA into hardware components (wires, transistors et cetera). Some models might result in subgraph homeomorphisms being found, while some models might not. A possible technique to remedy this is to attempt several different models simultaneously, using heuristics to estimate the likeliness of each model to result in a subgraph homeomorphism being found.

Even then, a subgraph homeomorphism may not be found even though a theoretical emulation exists: our methodology only looks for emulation of individual virtual components by individual concrete components and does not look for solutions where sets of virtual components are emulated by sets of virtual concrete components that may be composed of different component types. However, there might not be a solution to this: in general, an emulation mapping should map each possible function on the virtual FPGA to a semantically equivalent function on the concrete FPGA. Deciding whether two functions are semantically equivalent is an undecideable problem in general, only limited by the size of the FPGA. It is thus appriopriate to use a methodology that is most likely to find an emulation mapping where one exists, for which we provide a candidate with our methodology.

12 CONCLUSION

In this research, we aimed to investigate how to map a configuration for a virtual FPGA to one usable for a real-life concrete FPGA. One goal of such an emulation is allowing college students to learn about the FPGA compilation and synthesis process using a simple, easy-to-understand FPGA but still running their configurations on real hardware. Moreover, this problem fits within the greater research area of partial FPGA compilation: abtracting parts of an FPGA away such that they may be used for other purposes. We reduced this emulation problem to 'subgraph homeomorphism', an NP-complete graph problem. This is a problem for which several algorithms existed, but none met the space- and computational requirements needed for the scale of the inputs (graphs of the FPGAs). Because of this, we decided to adapt an algorithm such that it does meet the requirements. One of these algorithms was ndsh2: an algorithm that has minimum exponential space requirements. We adapted ndsh2, to a variant that only has polynomial space requirements (or even linear under some settings). To improve our chances of finding subgraph homeomorphisms, we added several optimisations: refusing paths using unnecessarily many FPGA resources, contraction, alldifferent pruning and various ways to order vertices in either graph. We found that each of these optimisations (except distance-based target graph vertex ordering) may reduce the time spent on finding homeomorphisms.

To answer our first research subquestion, we benchmarked the performance of this algorithm with optimal settings and extrapolate that cases similar to the business case will take approximately 6 days' time. Furthermore, it requires an approximately linear amount of memory for this computation. This is reasonable for an FPGA education environment, since this only has to be computed once before an emulation mapping is found on a machine with a realistic amount of memory. Unfortunately, our algorithm showed there is no subgraph homeomorphism from our virtual FPGA model to an ECP5 tile, or to blocks of up to 3-by-3 times. Since we were not able to find a subgraph homeomorphism between the FPGAs of the business case yet, we are unable to answer our second research subquestion: how many resources of the concrete FPGA are needed for each virtual logic case. We can, however, conclude a lower bound of 9 ECP5 tiles needed for a single virual logic cell[1]

[1] other layouts of 9 tiles or less that do not fit within a 3-by-3 grid might yield subgraph homeomorphisms for our business case; however, there are many of such layouts.

The algorithm we proposed is also applicable for graphs outside of the FPGA emulation domain. While the conclusions we make from our experiments are based on graphs representing FPGAs with their structure, our software toolbox with individually changeable settings allows for benchmarks on other graphs as well. Our software (or other software created with our methodology) can be used to establish the appriopriate configuration for any other domain reducable to subgraph homeomorphism.

Based on this algorithm, we establish a methodology to obtain a mapping for linear time emulation of virtual FPGAs on concrete FPGAs.

13 FUTURE RESEARCH

Through our research we encountered multiple opportunities for further research, both in our algorithm and in the general problem of FPGA emulation.

We improved Xiao's algorithm and optimized it for finding subgraph homeomorphisms between FPGA models. There are, however, techniques that we have not implemented but could yield performance gains of the algorithm.

- Since our algorithm is a form of search space exploration, exploration of different branches of the search tree could be performed in parallel. This has the potential of speeding up the algorithm by a factor of the number of computing cores used in parallel.
- Secondly, our algorithm could be improved by adding backmarking and backjumping [32]. This technique improves the pruner by recognising which addition of the partial mapping caused the pruner to kick-in, backtracking potentially several steps instead of one. This technique was already implemented in Glassgow [41], a subgraph isomorphism algorithm that is as of now not shown to be superceded by a better performing algorithm (See Appendix .1).
- The contraction optimisation has room for improvement. Whenever a duplicate edge appears due to contraction of some transistor or port, the algorithm attempts to find appropriate paths for the two edges independently. An optimisation would be to avoid mapping them with a set of paths if some mapping from those edges to the same set of paths has already been attempted before. Furthermore, we observe that contraction sometimes results in a performance deficit. If we can somehow find out which specific contracted vertices cause this behaviour, we can refrain from contracting them resulting in an overall better performance.
- The pruner currently has no knowledge of contracted vertices or limitations of mapping edges with contracted vertices to paths. Taking this into account in the pruner could result in some speedup.
- Our algorithm could take hierarchy into account. Whenever the source graph contains some graph pattern multiple times, it may be easier to find matches in the target graph for these repetitions after one instance is completely included in the

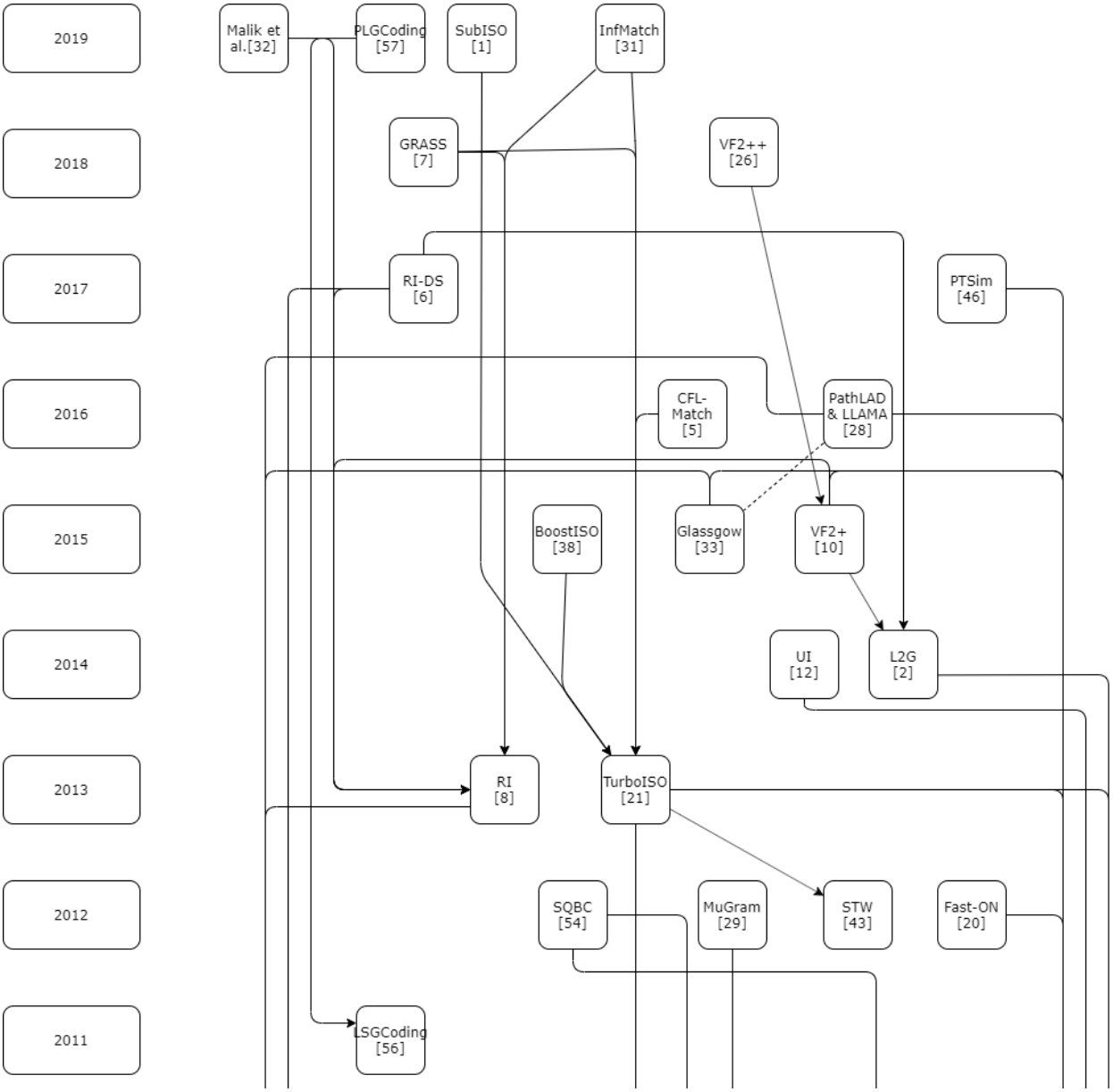
partial mapping.

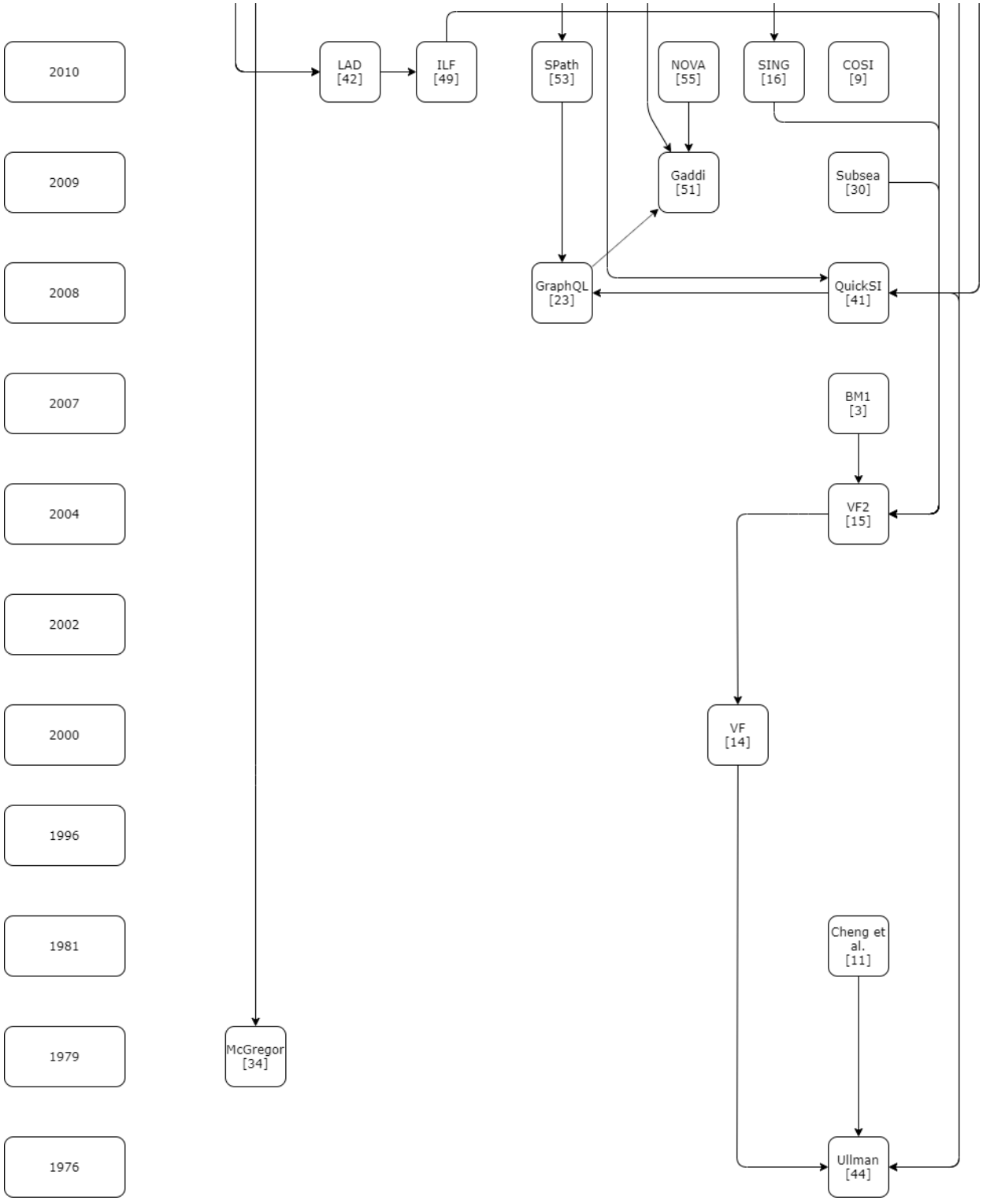
- Lastly, if one is able to retrieve information about the physical location of components of the concrete FPGA, this information can be used as heuristic for vertex orderings.

There are other areas of research possible as well. One of the problems of our methodology is that it requires a 1-to-1 mapping of physical components. If we build a repository of component structures along with structures that can emulate them, we could move away from subgraph homeomorphism to a more general variant that allows mapping groups of components on other groups. Lastly, we observe that, contrary to placement & routing algorithms, our algorithm performs best when the sizes of the two inputs are close together. Combining our research with research on placement & routing algorithms has the potential of resulting in an approach that works well with approximately equal input sizes and with widely different input sizes.

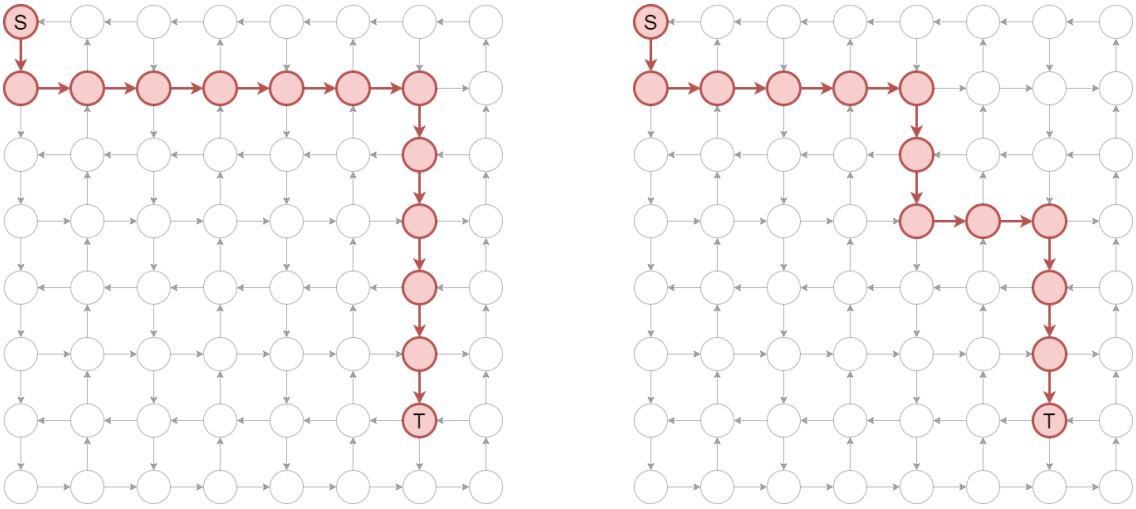
Appendices

.1 History of subgraph isomorphism algorithms



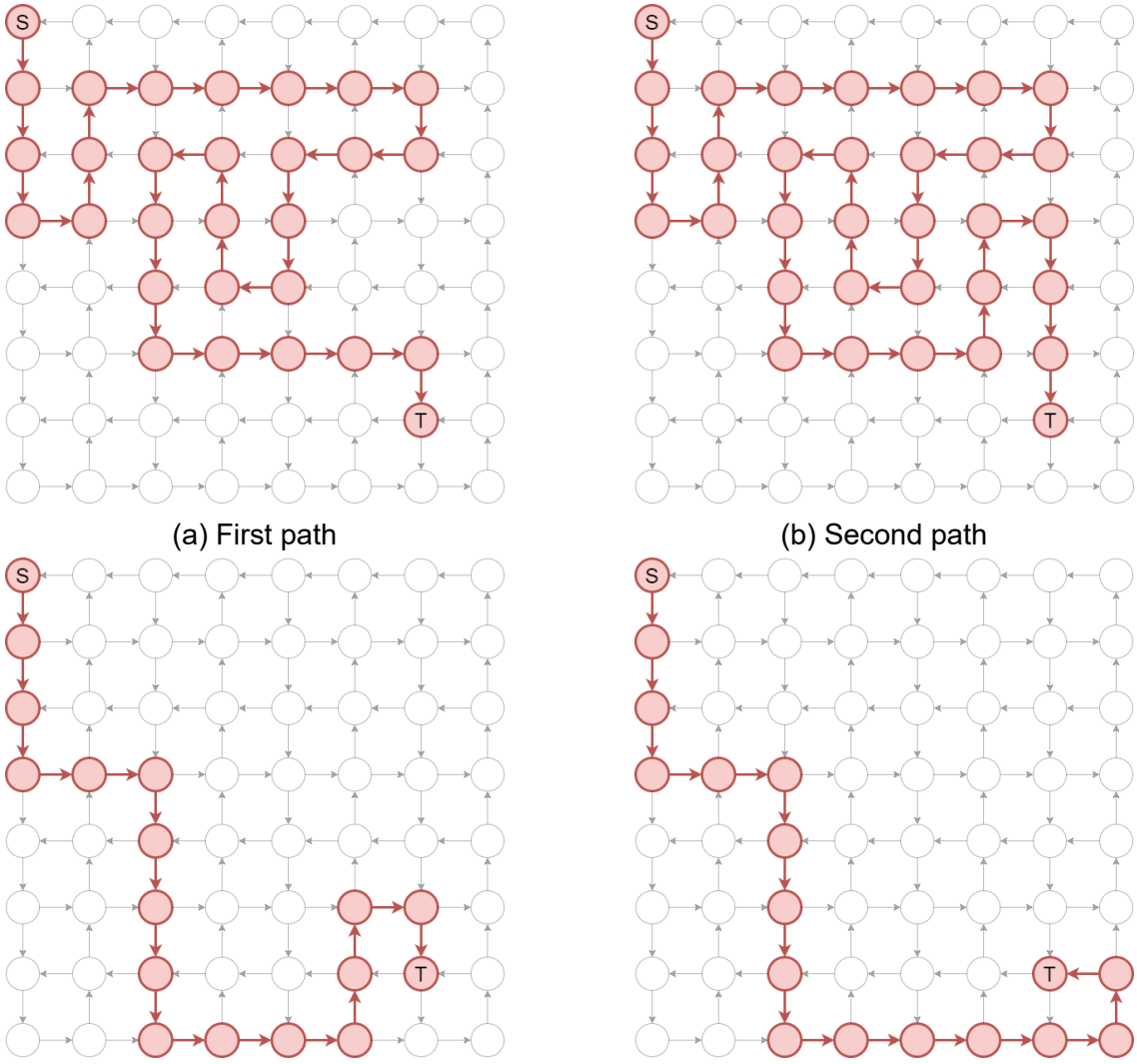

2010
2009
2008
2007
2004
2002
2000
1996
1981
1979
1976
LAD [42]
ILF [49]
SPath [53]
NOVA [55]
SING [16]
COSI [9]
Gaddi [51]
Subsea [30]
GraphQL [23]
QuickSI [41]
BM1 [3]
VF2 [15]
VF [14]
Cheng et al. [11]
McGregor [34]
Ullman [44]

.2 Examples of path iterators



(a) First path (b) Second path

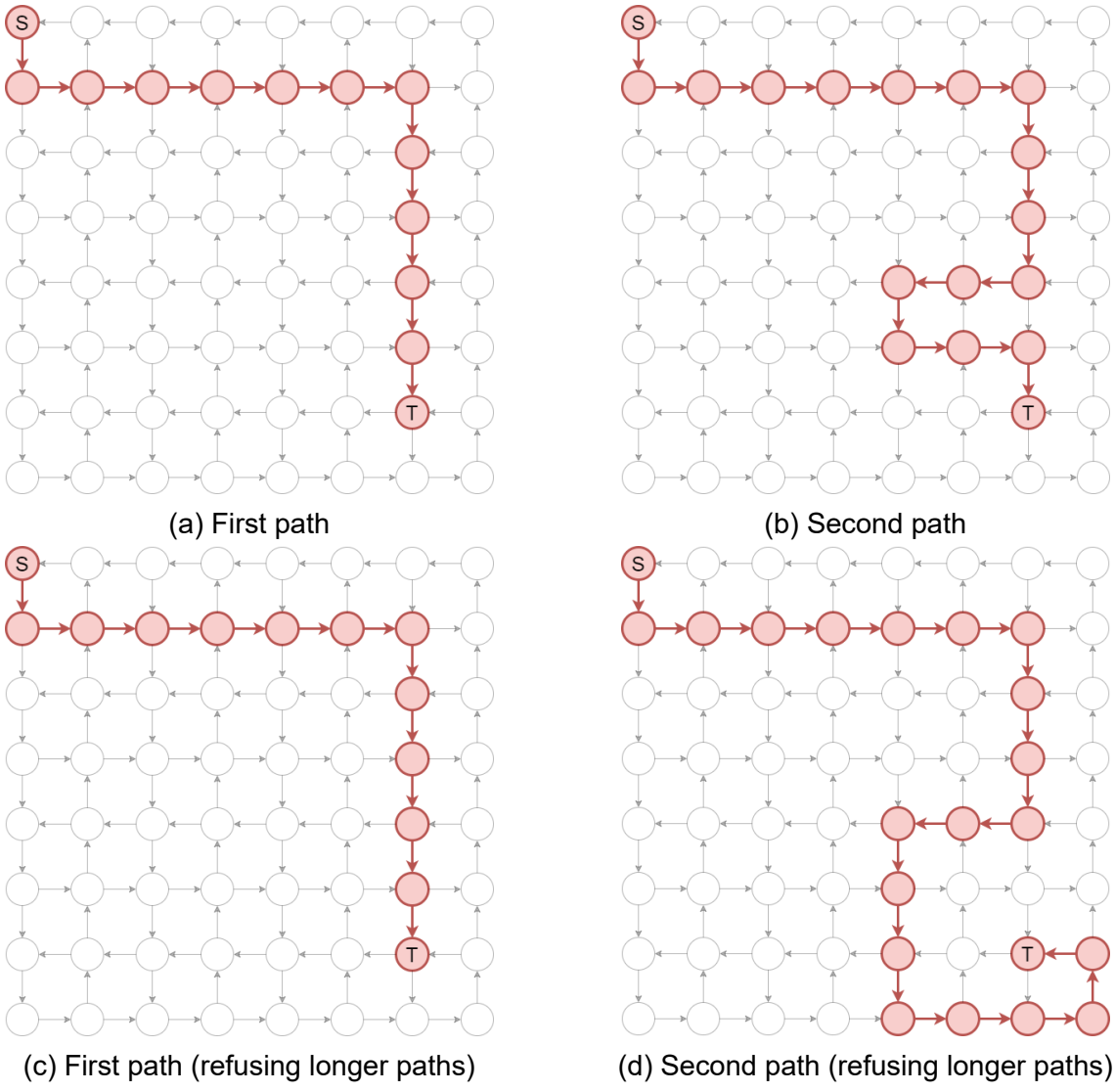
Figure 1: The first two iterations of the K-path path iterator in a square graph. This example is unaffected by “refusing longer paths”.



(a) First path (b) Second path

(c) First path (refusing longer paths) (d) Second path (refusing longer paths)

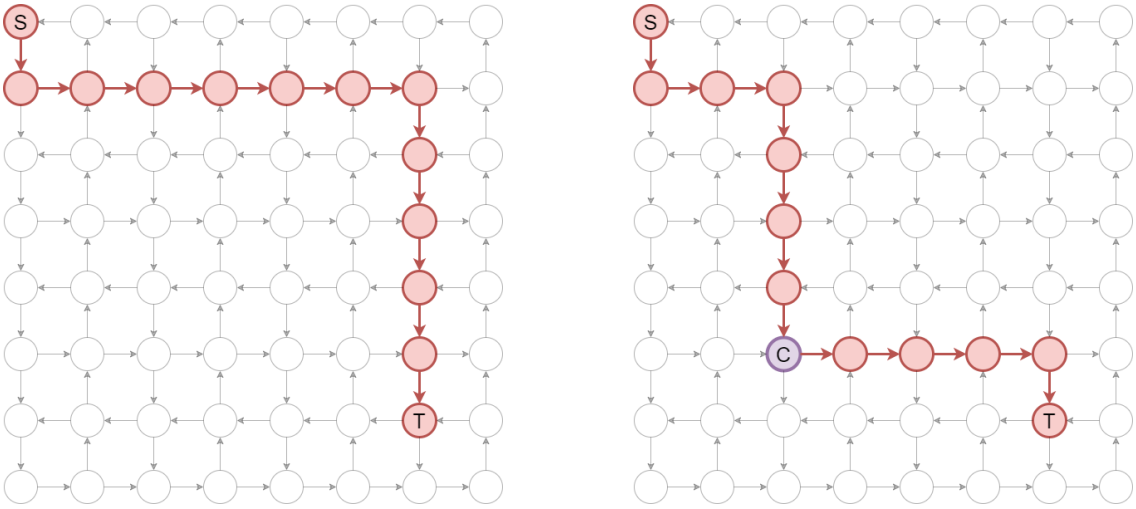
Figure 2: The first two iterations of the DFS path iterator in a square graph.



(a) First path (b) Second path

(c) First path (refusing longer paths) (d) Second path (refusing longer paths)

Figure 3: The first two iterations of the greedy DFS path iterator in a square graph.



(a) First path (0 control points)

(b) Second path (1 control point)

Figure 4: The first two iterations of the control point path iterator in a square graph. This example is unaffected by “refusing longer paths”.

.3 Proof: contraction preserves subgraph homeomorphism

Proof. Let $\textit{vcontract}_{S' \to S}$ and $\textit{econtract}_{S' \to S}$ be the (injective) vertex-to-vertex mapping and (injective) edge-to-path mapping from the contracted graph S_{cont} to S, respectively, and let $(\textit{vmap}_{S \to T}, \textit{emap}_{S \to T})$ be some vertex disjoint subgraph homeomorphism from S to T, respectively.

We then construct a vertex-to-vertex mapping $\textit{vmap}_{S' \to T}$ by mapping each vertex $v \in V_{S'}$ to $\textit{vmap}_{S \to T}(\textit{vcontract}_{S' \to S}(v))$. Recall from the definition of subgraph homeomorphism that $\forall s \in V_S . L_S(s) \subseteq L_T(\textit{vmap}_{StoT}(s))$. This holds for S and T since we assumed a subgraph homeomorphism existed between the two.

Since for every vertex $v \in V_{S'}$ we have $\textit{vcontract}_{S' \to S}(v) \in V_s$, it also holds that $\forall s' \in V_{S'} . L_S(\textit{vcontract}_{S' \to S}(s')) \subseteq L_T(\textit{vmap}_{S \to T}(\textit{vcontract}_{S' \to S}(s')))$. Moreover, since labels of remaining vertices are preserved through contraction, we have $\forall v \in V_{S'} . L_{S'}(v) = L_S(\textit{vcontract}_{S' \to S}(v))$. Therefore, it holds that: $\forall s' \in V_{S'} . L_{S'}(s') \subseteq L_T(\textit{vmap}_{S \to T}(\textit{vcontract}_{S' \to S}(s')))$. Therefore, this vertex-to-vertex mapping satisfies the first prequisite out of three for vertex disjoint subgraph homeomorphism.

We then construct an edge-to-path mapping $\textit{emap}_{S' \to T}$. For each edge $(u, v) \in E_{S'}$, we take the edges of the path $\textit{econtract}_{S' \to S}((u, v))$ and concatenate the edges of the corresponding paths in T to obtain a new path p'. We then add $((u, v), p')$ to $\textit{emap}_{S' \to T}$. Recall from the definition of subgraph homeomorphism that:

$$\forall (s_1, s_2) \in E_S. \quad \begin{array}{l} \textit{first}(\textit{emap}_{S \to T}(s_1, s_2)) = \textit{vmap}_{S \to T}(s_1) \wedge \\ \textit{last}(\textit{emap}_{S \to T}(s_1, s_2)) = \textit{vmap}_{S \to T}(s_2). \end{array}$$

Applying this formula to two consecutive edges allow us to conclude:

$$\forall (s_1, s_2), (s_2, s_3) \in E_S. \quad \begin{array}{l} \textit{first}(\textit{emap}_{S \to T}(s_1, s_2)) = \textit{vmap}_{S \to T}(s_1) \wedge \\ \textit{last}(\textit{emap}_{S \to T}(s_2, s_3)) = \textit{vmap}_{S \to T}(s_3). \end{array}$$

And similarly for more concatenated edges. Choosing the edge sequences such that each edge sequence in S corresponds to a single edge in S' gives us:

$$\forall (s'_1, s'_2) \in E_{S'}. \quad \begin{array}{l} \textit{first}(\textit{emap}_{S \to T}(\textit{firstEdge}(\textit{econtract}(s'_1, s'_2)))) = \textit{vmap}_{S \to T}(\textit{vcontract}(s_1)) \wedge \\ \textit{last}(\textit{emap}_{S \to T}(\textit{lastEdge}(\textit{econtract}(s'_1, s'_2)))) = \textit{vmap}_{S \to T}(\textit{vcontract}(s'_2)) \end{array}$$

which shows we satisfy the second prerequisite.

Lastly, we prove that performing a single contraction does not violate the third prerequisite

$$\forall p \in \textit{values}(emap_{S \to T}). \forall x \in \textit{intermediate}(p). \nexists p' \in (\textit{values}(\textit{emap}_{S \to T}) \backslash \{p\}). x \in \textit{intermediate}(p')$$

and induce that performing any number of contractions does not. Obviously, the base case holds since it is our assumption (i.e. S is vertex disjoint subgraph homeomorphic to T).

Let $s \in V_S$ have indegree 1 and outdegree 1, qualifying it for contraction. Let the edges be $(\textit{prec}(s), s)$ and $(s, \textit{succ}(s))$ where $\textit{prec}(s)$ may be equal to $\textit{succ}(s)$. We know that $\textit{emap}_{S \to T}(\textit{prec}(s), s)$ is internally vertex disjoint from all other paths in $\textit{values}(\textit{emap}_{S \to T})$, as well as $\textit{emap}_{S \to T}(u, \textit{succ}(u))$. They share at least an end vertex $\textit{vmap}_{S \to T}(s)$ and possibly $\textit{vmap}_{S \to T}(\textit{prec}(s))$ if $\textit{prec}(s) = \textit{succ}(s)$. We know that $\textit{vmap}_{S \to T}(s)$ is not the end vertex of another path since s has indegree- and outdegree 1, and not the intermediate vertex of another path since that is not permitted for subgraph homeomorphism. All in all, the combined paths contain intermediate vertices $\textit{intermediate}(\textit{emap}_{S \to T}(\textit{prec}(s), s)) \cup \textit{intermediate}(\textit{emap}_{S \to T}(s, \textit{succ}(s))) \cup \{\textit{vmap}_{S \to T}(s)\}$ that are not intermediate vertices of other paths in $\textit{values}(\textit{emap}_{S \to T})$.

When we contract this vertex, we get a new edge $(\textit{prec}(s), \textit{succ}(s))$ with start vertex $\textit{vmap}_{S \to T}(\textit{prec}(s))$, end vertex $\textit{vmap}_{S \to T}(\textit{succ}(u))$, and as intermediate vertices $\textit{intermediate}(\textit{emap}_{S \to T}(\textit{prec}(s), s)) \cup \textit{intermediate}(\textit{emap}_{S \to T}(s, \textit{succ}(s))) \cup \{\textit{vmap}_{S \to T}(s)\}$. The vertex set used in T remains exactly the same, thus does still not share intermediate vertices with other paths in $\textit{values}(\textit{emap}_{S \to T})$.

By induction, the third prerequisite holds. Since all prerequisites hold, S' is a vertex disjoint subgraph homeomorphism of T.

□

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. Brander, A.W., and Sinclair, M.C.: “A Comparative Study of k-Shortest Path Algorithms”. In: Performance Engineering of Computer and Telecommunications Systems: Proceedings of UKPEW’95, Liverpool John Moores University, UK. 5–6 September 1995. Ed. by M. Merabti, M. Carew, and F. Ball. London: Springer London, 1996, pp. 370–379. ISBN: 978-1-4471-1007-1. DOI: 10.1007/978-1-4471-1007-1_25. https://doi.org/10.1007/978-1-4471-1007-1_25.
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. 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)
21. 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)
22. Gouda, K., and Hassaan, M.: A fast algorithm for subgraph search problem. In: DE53–DE58 (2012)
23. Grohe, M., Kawarabayashi, K.-I., Marx, D., and Wollan, P.: Finding topological subgraphs is fixed-parameter tractable. In: pp. 479–488 (2011)
24. Han, W.-S., Lee, J., and Lee, J.-H.: Turbo<inf>ISO</inf>: 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)
25. Hauck, S., and DeHon, A.: Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2007)
26. 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)
27. Hershberger, J., Maxel, M., and Suri, S.: Finding the k Shortest Simple Paths: A New Algorithm and Its Implementation. ACM Trans. Algorithms 3(4), 45–es (2007)
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. Kondrak, G., and van Beek, P.: A theoretical evaluation of selected backtracking algorithms. Artificial Intelligence 89(1), 365–387 (1997)
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. Lingas, A., and Wahlen, M.: On Exact Complexity of Subgraph Homeomorphism. In: Cai, J.-Y., Cooper, S.B., and Zhu, H. (eds.) Theory and Applications of Models of Computation, pp. 256–261. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
38. 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)

39. 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)
40. Malík, J., Suchý, O., and Valla, T.: Efficient Implementation of Color Coding Algorithm for Subgraph Isomorphism Problem (2019)
41. McCreesh, C., and Prosser, P.: A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs (2015)
42. McGregor, J.: Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. Information Sciences 19(3), 229–250 (1979)
43. 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)
44. *Project Trellis*. SymbiFlow Team. GitHub, 2018.
45. Régin, J.-C.: Filtering algorithm for constraints of difference in CSPs. In: pp. 362–367. AAAI, Menlo Park, CA, United States (1994)
46. Ren, X., and Wang, J.: Exploiting Vertex Relationships in Speeding up Subgraph Isomorphism over Large Graphs. Proc. VLDB Endow. 8(5), 617–628 (2015)
47. 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)
48. 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)
49. 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)
50. Solnon, C.: AllDifferent-based filtering for subgraph isomorphism. Artificial Intelligence 174(12-13), 850–864 (2010)
51. 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)
52. Ullmann, J.R.: An Algorithm for Subgraph Isomorphism. J. ACM 23(1), 31–42 (1976)
53. Wolf, C.: Yosys Open SYnthesis Suite. (2016)
54. 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)
55. 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)
56. Yalla, P., and Kaps, J.-P.: Lightweight cryptography for FPGAs. In: pp. 225–230 (2009)
57. Yen, J.Y.: Finding the K Shortest Loopless Paths in a Network. Management Science 17(11), 712–716 (1971)
58. Yu, H., Lee, H., Lee, S., Kim, Y., and Lee, H.-M.: Recent advances in FPGA reverse engineering. Electronics (Switzerland) 7(10) (2018)
59. Zampelli, S., Deville, Y., and Solnon, C.: Solving subgraph isomorphism problems with constraint programming. Constraints 15(3), 327–353 (2010)
60. Zhang, S., Hu, M., and Yang, J.: TreePi: A novel graph indexing method. In: Proceedings - International Conference on Data Engineering, pp. 966–975 (2007)
61. 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)
62. 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)

63. Zhao, P., and Han, J.: On Graph Query Optimization in Large Networks. Proc. VLDB Endow. 3(1–2), 340–351 (2010)
64. 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)
65. 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 subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 5981 LNCS(PART 1), 140–154 (2010)
66. Zhu, L., and Song, Q.: A study of Laplacian spectra of graph for subgraph queries. In: pp. 1272–1277 (2011)
67. 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)