University of Twente
EEMCS
Formal Methods and Tools

FPGA-on-FPGA emulation using node disjoint subgraph homeomorphism

MSc Thesis

Pim van Leeuwen (s1602772)

Technical Supervisor
dr. W. Kuijper
Academic Supervisors
dr. ir. R. Langerak
H.H. Folmer MSc.

UNIVERSITY OF TWENTE.

nedap

Abstract

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

CONTENTS

1 Introduction **3**

2 Background **5**
2.1 FPGAs 5
2.1.1 Lookup tables 6
2.1.2 Registers 7
2.1.3 Logic cells 7
2.1.4 Routing 8
2.1.5 Pins 8
2.2 FPGA compilation 9
2.2.1 Literature 10
2.3 FPGA simulation 11
2.4 FPGA emulation 12
2.5 Graph theory 12
2.5.1 Literature 14

3 Objectives **18**

4 Models **19**

5 Algorithm **20**
5.1 Baseline: ndSHD2 20
5.2 How to choose vertex-vertex pairs 22
5.3 Source graph vertex order 22
5.4 Target graph vertex order 23
5.5 Path iteration 24
5.6 Optimisations 25
5.6.1 Refusing long paths 25
5.6.2 Runtime Pruning 26
5.6.3 Contraction 26

6 Pruning **29**

6.1 Domain filtering . . . 30
6.1.1 Labels and neighbours . . . 30
6.1.2 Free neighbours . . . 30
6.1.3 Reachability of matched vertices (M-filtering) . . . 31
6.1.4 Reachability of neighbourhood (N-filtering) . . . 31
6.2 Pruning methods . . . 32
6.2.1 ZeroDomain pruning . . . 32
6.2.2 AllDifferent pruning . . . 33
6.3 When to apply . . . 33
6.3.1 Runtime calculation . . . 33
6.3.2 Caching domains - incremental domain calculation . . . 33
6.3.3 Parallel calculation . . . 34

7 Use case: Lattic ECP5 35

8 Performance experiments 37

9 Conclusion 47

10 Discussion 48

11 Future Research 49

Appendices 50
.1 History of subgraph isomorphism algorithms . . . 51
.2 Examples of path iterators . . . 53
.3 Proof: contraction preserves verted disjoint subgraph homeomorphism . . . 57

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[20], cryptography[54, 42, 28], digital signal processing[19], protocol implementations[46] and other types of high-performance computing[30].

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[18]. Another option is to simulate FPGA programs instead. Simulation circumvents the place & route process. However, simulation software run on a CPU has to simulate each component sequentially for each CPU core, resulting in enormous performance loss.

Another technique is modelling the semantics of 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 2.3 and Figure 2.6. The trade-off here is that the hardware uses many resources for the simulation of each component. Because of this increased resource consumption, the performance of FPGA programs decreases as well.

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

We propose an education environment where students implement program logic and place & route manually in a simple, virtual FPGA. Performing the NP-complete place & route manually allows the rest of the compilation process to take linear time. This compilation process specifically compiles an FPGA-configuration meant for a virtual FPGA layout to a configuration compatible with a concrete one.

This way, students can circumvent the time-consuming compilation from HDLs to FPGAs, resulting in faster iterations and improved learning experiences. Furthermore, students can use this environment to learn the inherent difficulty of place & route before they use tools that encapsulate this process in a 'black box'. They will have to do this when they implement more complex structures in the virtual environment. We can achieve this emulation by calculating a mapping from the configuration of a virtual FPGA to the configuration of a real-life FPGA board (concrete FPGA). While the format of an FPGA configuration is usually kept secret by hardware vendors[24, 60, 4], some FPGA boards have been reverse engineered to reveal the underlying format[56]. Once this technique has been implemented and tested on those FPGAs, It can be put in practice in lab sessions of colleges and universities. Moreover, vendors can implement it for undisclosed FPGA designs and configuration formats as well for educational purposes as well.

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

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

2 BACKGROUND

2.1 FPGAs

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

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

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

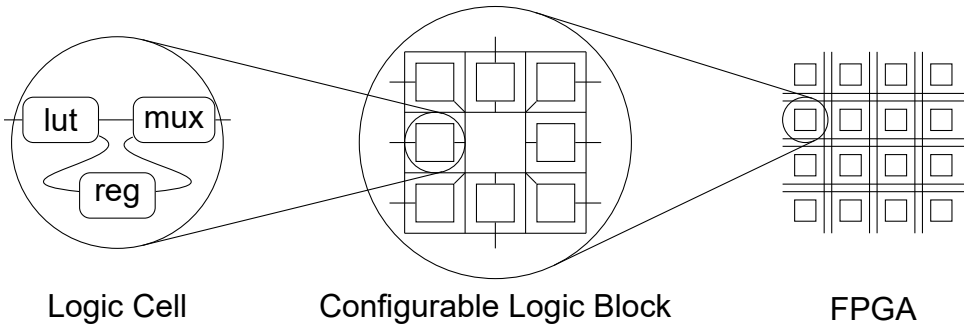


Figure 2.1: The hierarchy of a typical FPGA. A typical FPGA mostly consists of Configurable Logic Blocks (CLBs) and a CLB mostly consists of logic cells. The way logic cells and CLBs are connected may be different for each FPGA architecture.

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

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

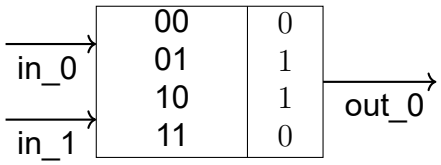


Figure 2.3: A Lookup Table (LUT) in which an XOR-operation is configured

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 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.

2.1.1 Lookup tables

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

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

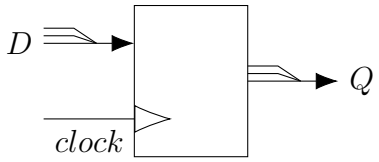


Figure 2.4: Traditional representation of a register. Note that input *D* and output *Q* may consist of multiple wires.

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.

2.1.2 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 2.4). Registers can store a small collection of bits, depending on the FPGA design. When a register's input $clock$ changes from 0 to 1, the contents of the register are replaced by the value of the input D, and the output Q takes this value. The output stays constant until the clock changes from 0 to 1 again when D has a different value.

FPGAs usually have 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.

2.1.3 Logic cells

A typical logic cell is a combination of one or more lookup tables, a register, and a MUX (a 3-input gate that outputs a copy of the first or second input, depending on the value of the third input). The contents of the lookup tables can be configured to perform any logic

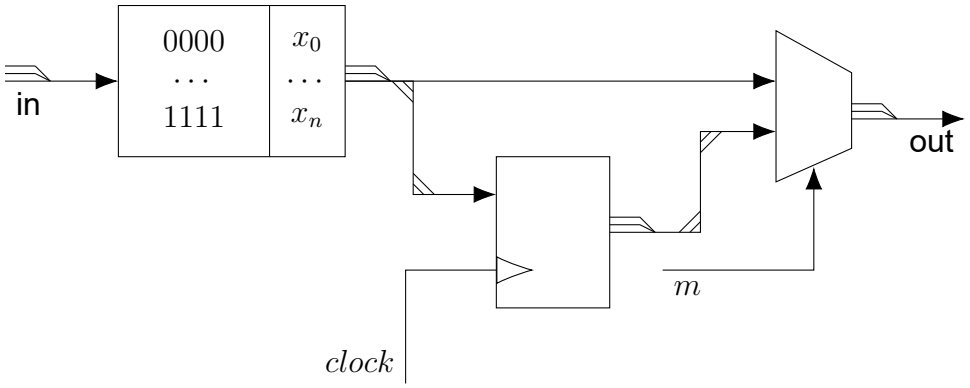


Figure 2.5: An n-input logic cell. x_k and m must be configured for any $0 \leq k \leq n$

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 2.5. 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 2.2) easier.

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

2.1.4 Routing

A single logic cell can only perform simple programs such as logic gates. The FPGA needs to connect different logic cells to perform any kind of logic operation, dependent on what program it needs to execute. The way logic cells and other modules are connected on a physical FPGA in a specific configuration is called the **routing** of an FPGA. On the most basic level, FPGAs perform routing with configurable 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.

2.1.5 Pins

To supply the FPGA with input and to allow it to provide output, an FPGA is equipped with a set of metal pins. Each of those pins is connected with a wire on the FPGA board. Each pin can be used for either input or output, depending on the configured program.

For example, an FPGA used to control an electric stepper motor will receive input with the requested motor speed and direction and will output signals to specific circuits that need to be activated to obtain that speed and direction.

2.2 FPGA compilation

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

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

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

The last step in the compilation process is **routing**. State-of-the-art FPGA compilation pipelines perform this step sequentially after placement[27]. 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 [35] 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.

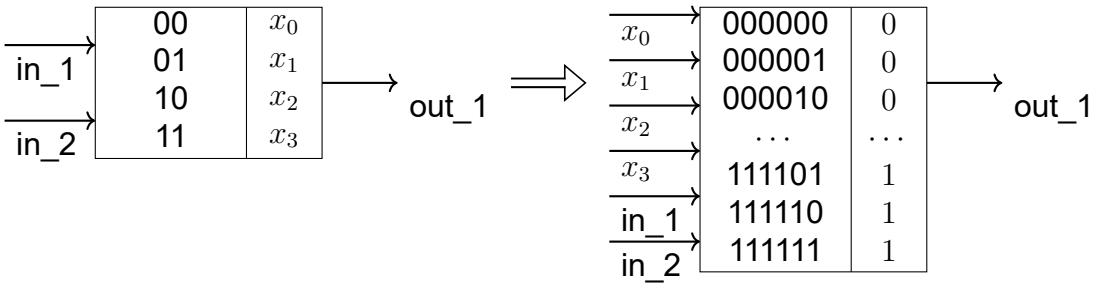


Figure 2.6: Simulation of an FPGA: the programming of the virtual FPGA is reflected in additional input of the actual FPGA. The value of x in the simulation is identical to the configuration of the program to be simulated.

! Note that place & route of an FPGA *program* as described here is a very similar problem to the placement of an unconfigured virtual FPGA. Place & route of an FPGA program has significant amounts of existing research, whilst placement of a virtual FPGA does not and is part of this research. The two problems are on different levels of abstraction: an FPGA program does not have the concept of unconfigured components or routing switches, while a virtual FPGA does.
Placement of FPGA programs using place & route could theoretically also work on a higher abstraction level, i.e. including unconfigured components. The problem with this approach is twofold: state-of-the art place & route pipelines perform these algorithms sequentially: they perform placement based on speed- and routeability heuristics without a guarantee that the placement can indeed be routed[27]. With the placement of FPGA programs, this is not a problem. Entire FPGAs, however, can have dense collections of components that can severely impact routeability. Secondly, it requires the virtual components to be compatible with the concrete components, which is not guaranteed.
Not using well-founded placement algorithms based on heuristics will undoubtedly have an adverse impact on the speed of our placement. However, we expect that we can use the hierarchical structure of an FPGA for significant improvements that could not be obtained with the placement of FPGA programs.

2.2.1 Literature

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

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

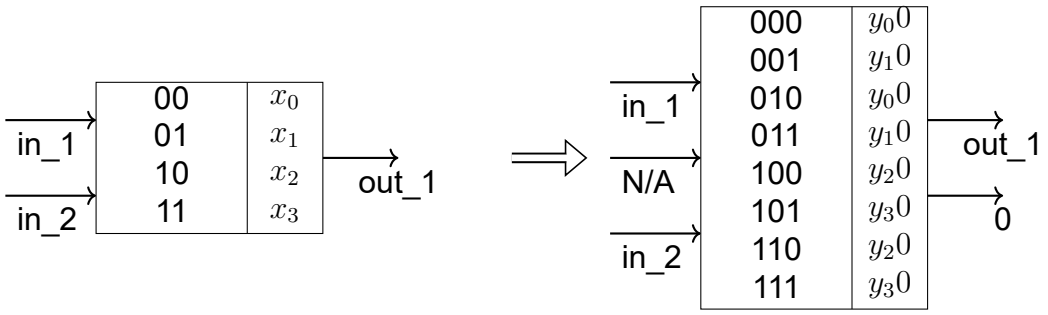


Figure 2.7: Emulation of an FPGA: the programming of the virtual FPGA is reflected in the programming of the actual FPGA. In this example, y_i takes the value of x_i. Any extra inputs and outputs are unused.

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

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

2.3 FPGA simulation

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

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

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

2.4 FPGA emulation

Instead, our research entails finding out how we can perform *emulation*. With this, we mean the execution of a program on a different FPGA *such that no extra input is required.* Instead, the emulator software inspects the program for the virtual FPGA and generates what program on the concrete FPGA would have the same semantics given the same inputs as the source FPGA. Figure 2.7 shows the emulation counterpart of Figure 2.6. Since the target FPGA has the same LUT type in this example, the emulation converts the LUT configuration to an identical configuration on the target FPGA. We aim to find out how we can perform this kind of emulation with any program with any fixed source- and destination FPGA whenever such emulation is possible.

2.5 Graph theory

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

The graph theoretic name for finding these embeddings is subgraph isomorphism. It is an NP-complete problem[**cook**] 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. This means that the embedding consists of a vertex-to-vertex mapping and an edge-to-path mapping.

> **Definition 2.5.1** (graph)**.** A graph is a tuple (V, E, L) where V is a set of vertices, E is a multiset of directed edges such that each edge $e \in E \rightarrow e \in (V \times V)$ and $L : (V \rightarrow \mathbb{P}(\lambda))$ is a multilabelling function where λ is a finite alphabet.
>
> **Definition 2.5.2** ((vertex disjoint) subgraph homeomorphism)**.** Let P the set of all loopless paths in G_2, let $first(p)$ be the first vertex of a path p, let $last(p)$ be the last

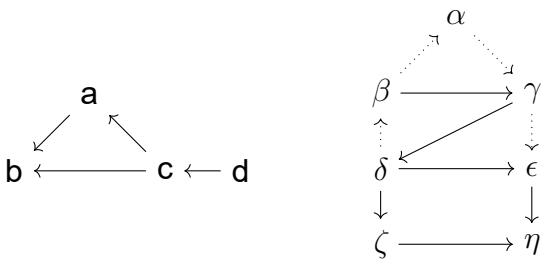


Figure 2.8: Two graphs G_1 and G_2. G_1 is (node disjoint) subgraph homeomorphic to G_2 with the mapping $\{(a,\epsilon),(b,\eta),(c,\delta),(d,\beta),(a \to b, \epsilon \to \eta),(c \to a, \delta \to \epsilon),(c \to b, \delta \to \zeta \to \eta),(d \to c, \beta \to \gamma \to \delta)\}$. Other homeomorphisms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

vertex of a path p and let $intermediate(p)$ be all other vertices of a path p. Then, a vertex disjoint subgraph homeomorphism from G_1 to G_2 is a tuple (f, q) where $f \subseteq (V_1 \mapsto V_2)$ and $q \subseteq (E_1 \mapsto P)$ are both injective functions such that:

1. $\forall u \in V_1.L_1(u) \subseteq L_2(f(u))$
   i.e. vertices are mapped to vertices that have at least the same label set.
2. $\forall (u_1,u_2) \in E_1.\textit{first}(q(u_1,u_2)) = f(u_1) \land \textit{last}(q(u_1,u_2)) = f(u_2)$
   i.e. each edge in G_1 is mapped to a path in G_2.
3. $\forall p \in \textit{values}(q).\forall x \in \textit{intermediate}(p).\ \nexists p' \in (\textit{values}(q) \setminus \{p\}).x \in \textit{intermediate}(p')$
   i.e. these paths are 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[2][3]. The vertex-to-vertex mapping f describes how to obtain configurations for the concrete FPGA and the edge-to-path mapping q 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.

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

[3]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.

2.5.1 Literature

We performed literature research to obtain existing subgraph isomorphism and subgraph homeomorphism algorithms, the results of which are shown in Appendix .1. We searched the Scopus, ACM and IEEE libraries for the terms “subgraph isomorphism”, “subgraph matching”, “subgraph homeomorphism” and “topological minors”. We examined matching papers and selected those that solve exact (not approximate) subgraph isomorphism or homeomorphism. We then iteratively added other algorithms from performance comparisons and citations. In the chart, an arrow implies that the paper of the newer algorithm or a survey paper showed that the algorithm at the source of the arrow performed better (i.e. lower mean time to find a matching) than the algorithm at the target of the arrow. A line between algorithms without arrowhead means a paper showed that the algorithms perform equally well.

Subgraph isomorhism

Malik et al.[38] simplify subgraph isomorphism by repeatedly randomly simplifying the target graph using a colouring.

PLGCoding[65], based on LSGCoding[64] uses the length of the shortest path and “Laplacian spectra” to effectively index the target graph and search for subgraphs. However, the invariants that these methods use to solve subgraph isomorphism do not necessarily hold for subgraph homeomorphism.

subISO[1] divides the target graph into subregions based on a pivot vertex in the pattern graph such that the number- and size of subregions is minimal. It then uses Ullmann[50] to search these subregions for the subgraph. Similarly, InfMatch[37] uses heuristics to select a node in the target graph and selects subregions in the target graph from that node to search in. PTSim[53] also applies graph partitioning, continuing with a different subgraph isomorphism algorithm on the resulting partitions, but only after first removing every edge from the target graph if the combination of labels of its source and target does not occur in the pattern graph. COSI[11] uses partitioning of graphs in cloud networks to find subgraph queries in social network graphs. Afterwards, it uses L2G’s algorithm[2].

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

VF2++[29] is an adaptation of VF2+[12], an improvement on VF2[16] which in its turn is an improvement of VF[15], 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[32] is a variation upon VF2 that allows multiple labels on each node. Since VF2++ has not been compared with RI-DS[6], it is

unknown whether a fixed node order is a more promising technique than a dynamic node order.

RI[8] is similarly to VF2++ a variant of DFS. It assigns a variable to each node in the pattern graph with the domain of possible matches in the target graph. In its DFS, it uses a *dynamic* ordering of nodes. A node in the target graph is matched next if it has many neighbours in the existing partial matching, many neighbours of neighbours in the existing partial matching or else a large degree. With this, RI aims to maximize the number of verifiable constraints. RI-DS[6] improves upon this by implementing a cache that checks label compatibility for nodes quicker. Rudolf[45] proposes a querying method to access graph data in a subgraph isomorphism problem within a constraint satisfaction problem context but does not provide an algorithm.

Similarly to RI, LAD[48] solves subgraph isomorphism using constraints. It then applies AllDifferent filtering (using an algorithm by Régin[43]) during constraint solving/DFS to reduce the domain as much as possible. This filtering removes every u from the domain of v whenever an assignment of v to u would result in an empty domain for some variable. McGregor[40] performs this as well but with a different AllDifferent algorithm. PathLAD[31] 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[31] and BM1[3] are portfolio techniques. Based on heuristics, they pick an approach from a collection that they expect to perform best. LLAMA picks from a collection of different algorithms whilst BM1 picks from different pruning settings for VF2.

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

Glassgow[39] 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[23] extracts subregions from the target graph by finding instances of a com-

pressed 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[62] takes cliques into account when searching for subgraphs: any node in a maximal clique in the pattern graph needs to be matched with a node in a maximal clique in the target graph that is at least as large.

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

GraphQL[25] 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 [57] 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[63] 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[58] and Gaddi[59] use the distances between node pairs to index graphs. Sing[17] improves upon this by indexing the graph on the fly during the search process. SPath[61] also uses indexing techniques on trees and subgraphs to speed up the search for a complete mapping.

Subsea[36] 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[47] makes use of a set of spanning entries to combine tree search with normal DFS.

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

Ullmann[50] is often used as a baseline algorithm when testing subgraph isomorphism algorithms. It performs DFS using nodes with the highest degree in the source graph with random nodes in the target graph. L2G[2] improves upon this by selecting unassigned nodes from the target graph that are connected to the partial matching first. Fast-ON[21] improves upon Ullmann's algorithm by ordering the pattern graph vertices by degree and taking labels into account. UI[14] improves upon Ullmann's algorithm[50] by ordering the vertices of the pattern graph by a 'subdegree' measure in descending order and by breaking ties by choosing the node with the highest closed-neighbourhood clustering value.

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

Subgraph Homeomorphism

Lingas et al.[34] present an algorithm for subgraph homeomorphism under the assumption the vertex placement is fixed. For general subgraph homeomorphism they suggest trying their algorithm on each possible vertex matching.

Xiao et al.[52] present an algorithm for subgraph homeomorphism when the length of the intermediate paths is in a limited range[4]. Since our path lengths can be $[1, -\infty)$, we would need to find an alternate solution.

Grohe et al.[22] 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.[33] present some ways to reduce the graphs in a subgraph homeomorphism problem. These reductions are, however, not applicable to FPGA graphs.

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

3 OBJECTIVES

This research 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.

4 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 2.5.1. 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`, 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 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.

5 ALGORITHM

5.1 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'[52] and Lingas algorithm[34]. We choose to adapt Xiao's algorithm ndSHD2 as baseline over Lingas because it is easier to understand (thus easier to improve, saving valuable research time) and includes methods for vertex-on-vertex mapping. 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**.

Algorithm 1: ndSHD2

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

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

```
if s is dead state then
    return false;
end
else if s is complete then
    return true;
end
found ⟵ false
while !found ∧ ∃ valid node/edge-path mapping pair do
    m ⟵ GetNextNodePair() // m ⟵ GetNextEdgePathPair();
    s' ⟵ BackupState(s);
    NodeMap_s ⟵ NodeMap_s ∪ {m} // EdgePathMap_s ⟵ EdgePathMap_s ∪ {m};
    Refine(M, R);
    found ⟵ ndSHD2(s, M, R);
    if found then
        return true;
    end
    s ⟵ RecoverState(s');
end
return false;
```

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 E_{source} to the path set of G_{target} 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 has been matched, the algorithm finds a random path between the mapped edge source and target in the target graph. The algorithm prioritizes extending the matching with an edge-path pair over extending it with a vertex-vertex pair. Whenever no such path or vertex exists, the algorithm undoes (backtracks) the last matching step taken and tries a different

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

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 $Refine(M, R)$ call. This method filters out future path candidates from a path storage that have an already mapped target graph vertex as intermediate vertex. This way, the resulting mapping is guaranteed to be vertex disjoint.

The algorithm as specified in Xiao's paper leaves out some specific ordering details- we will assume that they used random ordering. We specify which vertex order we use in Sections 5.3, 5.4. We will explain what order of paths we use in Section **??**.

5.2 How to choose vertex-vertex pairs

There are different ways to choose which vertex-vertex pair to add to the partial mapping first, and which only to try 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 a source graph vertex heuristic that is precomputed and provide 2 heuristics for target graph vertices, one of which is precomputed and one of which is calculated run-time.

5.3 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 $s_1 \prec s_2$ 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.

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 we extract that the best performing algorithm that adheres to partial mapping search for subgraph isomorphism is RI-DS. In our literature study, we found no evidence a faster algorithm exists.

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

> **Definition 5.3.1** (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 5.3.2** (neighbour set)**.** If some $v \in V$, then ***neighbours***$(v) :=$ ***succ***$(v) \cup$ ***pred***(v).

The algorithm RI-DS obtains a variable ordering using a greedy algorithm called "GreatestConstrainedFirst". This algorithm starts with an ordering μ containing only the source graph vertex with the highest degree, i.e. $v \in V_1$ where $|$***neighbours***$(v)|$ is highest. Then, each source graph vertex not yet in the order is assigned three scores N_1, N_2 and N_3. $N_1(s_x)$ is the number of neighbours of s_x that are already in the order (i.e. $|\{s_y | s_y \in$ ***neighbours***$(s_x) \wedge s_y \in \mu\}|$). $N_2(s_x)$ is the number of neighbours of s_x that are not in the order themselves, but do have a neighbour in the order (i.e. $|\{s_y | s_y \in$ ***neighbours***$(s_x) \wedge s_y \notin \mu \wedge |$***neighbours***$(s_y) \cap \mu| > 0\}|$). $N_3(s_x)$ is the number of all remaining neighbours of s_x (i.e. $|\{s_y | s_y \in$ ***neighbours***$(s_x) \wedge s_y \notin \mu \wedge |$***neighbours***$(s_y) \cap \mu| = 0\}|$). It selects each vertex v of which $N_1(v)$ 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 order, and the process repeats. This continues until every vertex has been added to the order.

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.

5.4 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[39]. 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, x are chosen using the following metric, choosing vertices with lower metric values first:

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

Another option is to take the current partial mapping into account. Using a degree-based or random target graph vertex order will result in consectively chosen target graph vertices being spread roughly evenly throughout graph. Since we optimise our algorithm for large target graphs, chosen vertices at arbitrary locations will likely result in long paths necessary to reach them, and thus in unneccessary resource usage. 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_x, 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[2] space usage (we implement this with- and without caching). Formally, when some source graph vertex s_x needs to be matched, t_x are chosen using the following metric, choosing vertices with lower metric values first:

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

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

[2] $O(|V_s| * |V_t|)$

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

Table 5.1: Worst-case space complexity of each path iteration strategy. The computational requirements of each method change in different ways for subsequent calls.

5.5 Path iteration

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

- **K-path -** Try all loopless paths from shortest to longest, avoiding unusable vertices in the existing partial mapping. We use Yen's algorithm[55] for this. Faster and more recent algorithms exist[26, 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 -** Search for paths using greedy depth first search, choosing the direction closest to the goal vertex first and avoiding unusable vertices in the existing partial mapping.
- **Control point -** Select increasingly many 'control points' (from 0 to $|V|$) randomly in the target graph that must be in the path in a specific order, then connecting them by a 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.[3]

These methods provide for each two vertices in a graph each path connecting them exactly once. The space requirements for each method are shown in Table 5.1, and examples of paths returned by them are shown in Appendix 2.

[3]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.

5.6 Optimisations

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

5.6.1 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 p is "unnecessarily long" iff:

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

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

5.6.2 Runtime Pruning

Some subgraph isomorphism algorithms[16, 39] prune the search space during the search using some detection method of dead search paths. In Chapter 6 we will elaborate on the different techniques we implement for subgraph homeomorphism.

5.6.3 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 and replacing them with a single edge, as illustrated in Figure 5.1. 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 loops. If a node disjoint subgraph homeomorphism exists from S to some target graph T, then there also exists a vertex disjoint 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.

However, not every subgraph homeomorphism between S' and T corresponds to a subgraph homeomorphism between S and T. For example, an edge resulting from contraction of a vertex with label set $\{$"*red*"$\}$ and a vertex with label set $\{$"*blue*", "*green*"$\}$ should only be mapped to a path in T whose intermediate vertices contain one with a label set at least

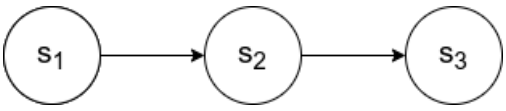
containing {"*red*"} and one with a label set at least containing {"*blue*", "*green*"} in that order. Only then will we have vertex candidates for contracted vertices in S after we found a subgraph homeomorphism between S' and T. When we contract vertices, therefore, we store the label sets of contracted vertices in order for each resulting edge. An example of contraction applied to a source graph with labels is shown in Figure 5.2.

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 $e \in E_{cont}$ with the same source- and target vertex we determine the set of label set sequences that their found mapped paths in T support. For edges that have no matches yet (initially each of them), we assume their path is compatible with each of those sequences. We then reverse this map to one that indicates for each label set sequence which paths are compatible. We then use AllDifferent (see Section 6.2.2) to determine whether each label set sequence can be mapped to a different path in T. If this is the case, we continue the partial mapping search; otherwise we backtrack and try a different edge.

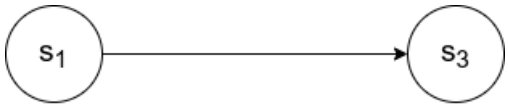
This optimisation introduces some delay since the algorithm needs to check compatibility between label set sequences and paths, and solve an all-different constraint. However, it also saves time since it reduces the source graph's size.

Contraction for undirected graphs

Contraction can also 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 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 $M(v)$.



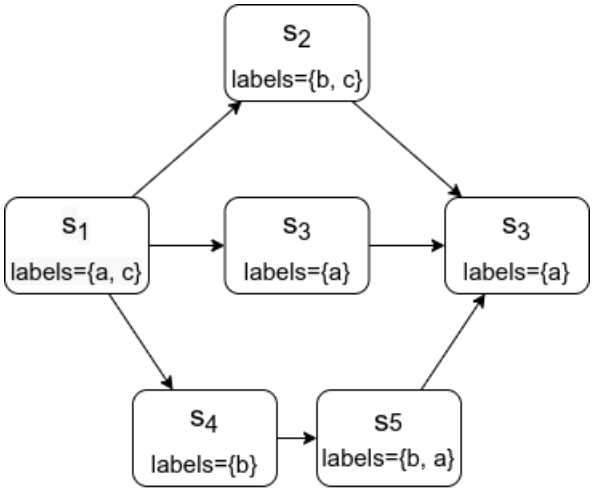
(a) Part of a source graph that may be contracted



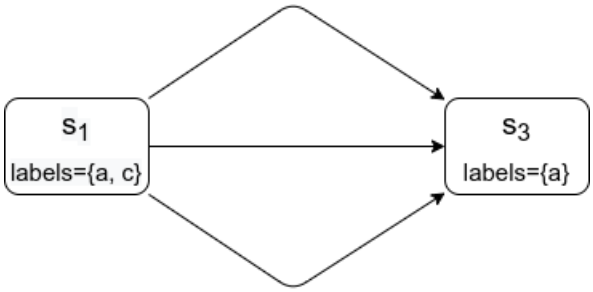
Edge	Contracted
(s_1, s_3)	{[∅]}

(b) The result of contraction

Figure 5.1: 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.



(a) A source graph before contraction



Edge	Contracted
(s_1, s_3)	{[{b, c}], [{a}], [{b}, {b, a}]}

(b) The same source graph after contraction.

Figure 5.2: 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 PRUNING

During the search for a complete matching, during which the algorithm only has *partial* matchings, the algorithm will often explore branches of the search tree that will not eventually lead to a homeomorphism. We will call these branches *dead branches*. Exploring an entire dead branch may be costly: its size is 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.

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

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

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

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

6.1 Domain filtering

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

1. If at least one homeomorphism can be found from this search branch, then for some homeomorphism that can be found from this search branch with vertex-on-vertex matching M_V it holds that $\forall(s \to t) \in M_V.t \in \textit{domain}(s)$.
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).

6.1.1 Labels and neighbours

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

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

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

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

6.1.2 Free neighbours

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

Definition 6.1.2 (compatibility under free neighbours constraint)**.** If M is the current partial matching, then:
$\textit{compatible}_{\textit{FN}}(v,u) := \textit{compatible}_{\textit{LABEL}}(v,u) \wedge C_{\textit{indegree}}(v,u) \wedge C_{\textit{outdegree}}(v,u)$
where:
$C_{\textit{indegree}}(u,v) := |\{v' \in \textit{pred}(v).v' \notin M\}| \geq |\{u' \in \textit{pred}(u).u' \notin M\}$
$C_{\textit{outdegree}}(u,v) := |\{v' \in \textit{succ}(v).v' \notin M\}| \geq |\{u' \in \textit{succ}(u).u' \notin M\}$

This method satisfies criterium since

6.1.3 Reachability of matched vertices (M-filtering)

A strong (and computationally very expensive) method of filtering the domain we will use[3] for some source graph vertex u and some target graph vertex v in addition to label- indegree and outdegree compability is to check that v can reach the mapped vertices of successors of u and that v can be reached from the mapped vertices of predecessors of u, i.e. that candidate paths exist for the upcoming edge-path matching attempts. This pruning method does not check whether a set of paths exist that is vertex disjoint since that is the task of the main algorithm: it merely checks the existence of paths. If no path exists, then no vertex disjoint path exists and pruning is required. An example of a domain filtered by reachability can be found in Figure 6.1. Formally:

Definition 6.1.3 (compatibility under reachability constraint)**.** If M is the current partial mapping and P is the set of paths in the target graph, then:
$\textit{compatible}_{\textit{REACH}}(v,u) := \textit{compatible}_{\textit{FN}}(v,u) \wedge C_{\textit{inr}}(v,u) \wedge C_{\textit{outr}}(v,u)$
where:

$$C_{\textit{inr}}(u,v) := \forall u' \in (\textit{pred}(u) \cap M).\exists p \in P.\quad \textit{first}(p) = M(u') \wedge \textit{last}(p) = v \wedge \nexists i \in \textit{intermediate}(p).i \in M$$

$$C_{\textit{outr}}(u,v) := \forall u' \in (\textit{succ}(u) \cap M).\exists p \in P.\quad \textit{first}(p) = v \wedge \textit{last}(p) = M(u') \wedge \nexists i \in \textit{intermediate}(p).i \in M$$

6.1.4 Reachability of neighbourhood (N-filtering)

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

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

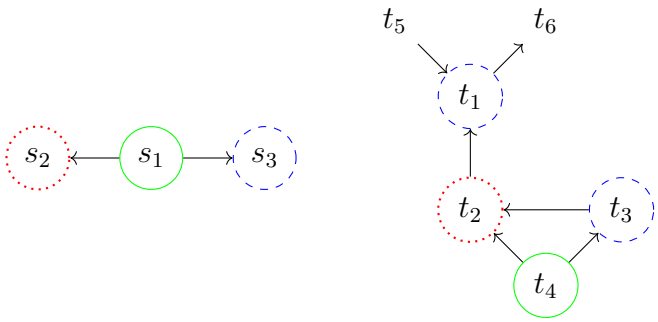


Figure 6.1: After vertex placements $s_1 \to t_4$ and $s_2 \to t_2$, the domain for vertex s_3 would normally be $\{t_1, t_3\}$. However, by checking for reachability from t_4 we can reduce this to $\{t_3\}$. The numbers represent the matching order and the circle styles represent labels.

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

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

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

6.2 Pruning methods

6.2.1 ZeroDomain pruning

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

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

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

6.2.2 AllDifferent pruning

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

6.3 When to apply

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

6.3.1 Runtime calculation

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

6.3.2 Caching domains - incremental domain calculation

Another option is to cache the domain of each variable and update it based on the current partial matching. This saves valuable time but requires quadratic space[4] (for each source graph vertex $\in |V_{source}|$ it needs to store a domain of average size $O(|V_{target}|)$). If the reachability pruning method is used, paths need to be cached as well. That is, for each edge $\in E_{source}$ we need to store a path of $O(|V_{target}|)$ vertices.

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

6.3.3 Parallel calculation

Finally, we can perform the domain filtering and procedure in a 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.

7 USE CASE: LATTIC ECP5

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

The structure of an ECP5 FPGA is shown in Figure **??**. 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[**todo**] to obtain graphs of both an individual tile and of collections of adjacent tiles.

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

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

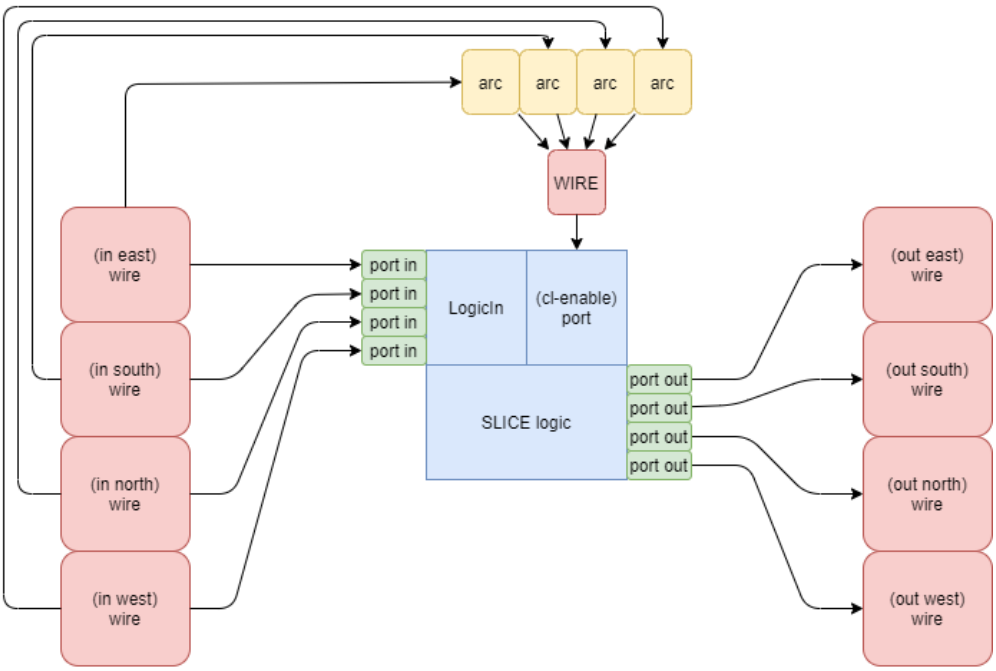


Figure 7.1: A graph model of a single cell of the virtual FPGA aimed to emulate on the Lattice ECP5

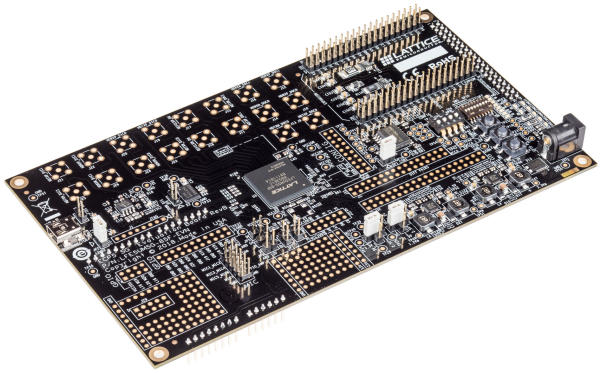


Figure 7.2: Evaluation board of the LFE5UM5G-85F FPGA: a variant of the ECP5.[1]

8 PERFORMANCE EXPERIMENTS

Using a machine with an AMD Ryzen 5 3600 CPU @ 3.7GHz and sufficient 3200MHz memory we conduct several experiments with different inputs and setting. The results are shown in Tables 8.2 through 8.6.

For source graph-target graph pairs without a present vertex disjoint subgraph homeomorphism we consistently use the G(n, M) Erdős–Rényi random graph model.

Since vertex disjoint subgraph homeomorphisms are rare in random graphs, we construct source graph-target graph pairs with a present vertex disjoint subgraph homeomorphism a different way. First, wegenerating a random source graph using the use the G(n, M) Erdős–Rényi random graph model. Then we add vertices: with a 50% probability we intersect a random existing edge with each new vertex and with 50% probability we add it outside the existing graph. Then we add edges: while the vertices added outside the source graph have less-than-average degree we add uniformly sampled edges that include them; afterwards we add uniformly sampled random edges.

Firstly, we compare the different methods of path iteration. The time taken to explore the entire search tree (without homeomorphism) gives an indication of the overhead introduced by a pathiterator, not taking heuristic value into account. This comparison is shown in Figure 8.3. The time taken for each path iteration method to find a homeomorphism where one exists is shown in Figure **??**. Here, the delay introduced caused by overhead may be offset by heuristic value.

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

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

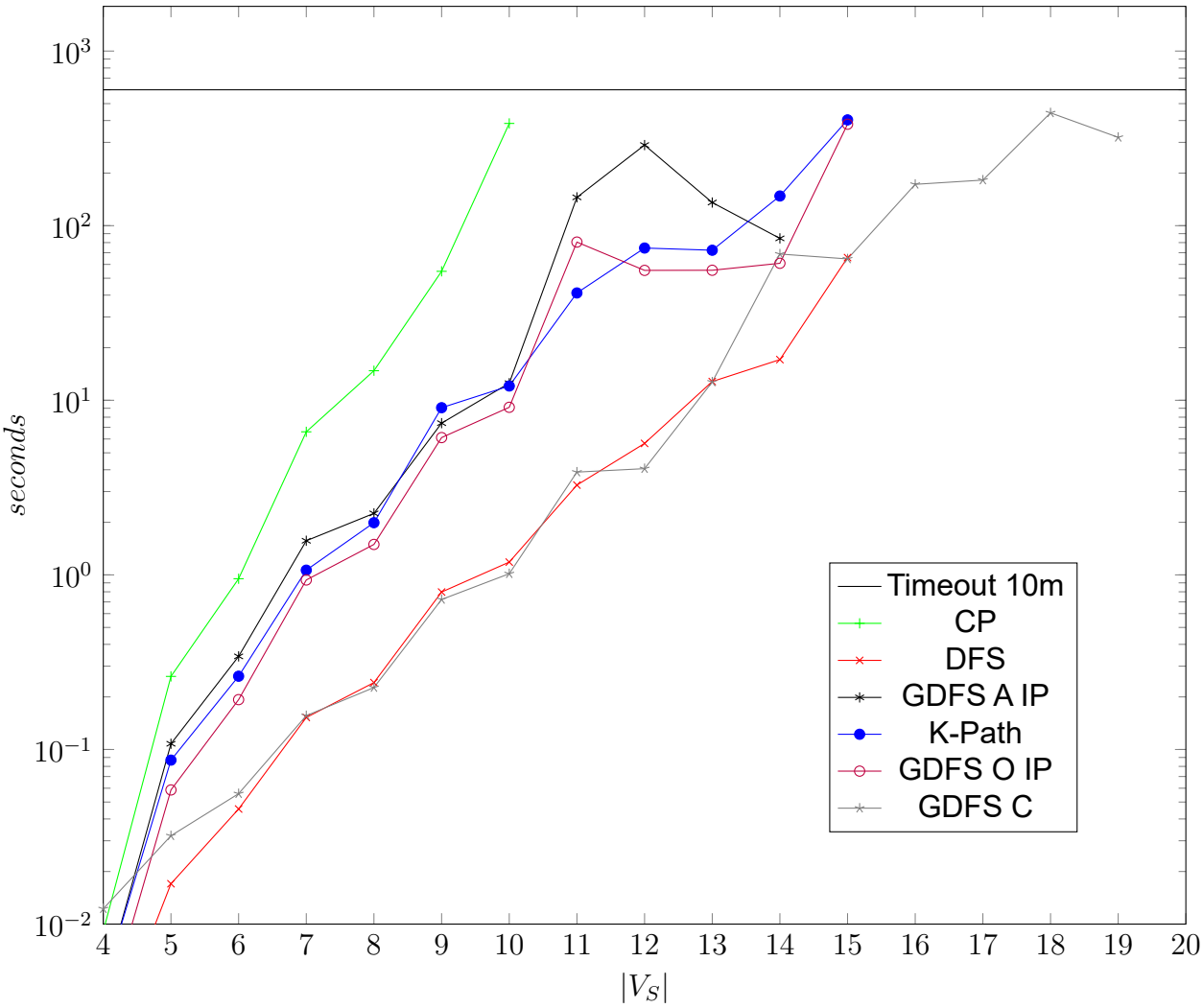


Figure 8.1: A comparison of the speed of path iterators on directed graphs without a vertex disjoint subgraph homeomorphism. The time taken is measured of exploring the entire search tree without pruning or contraction and "refuse longer paths" enabled. The source graph has average degree 3.0 and the random target graph has 50% more vertices and average degree 4.0.

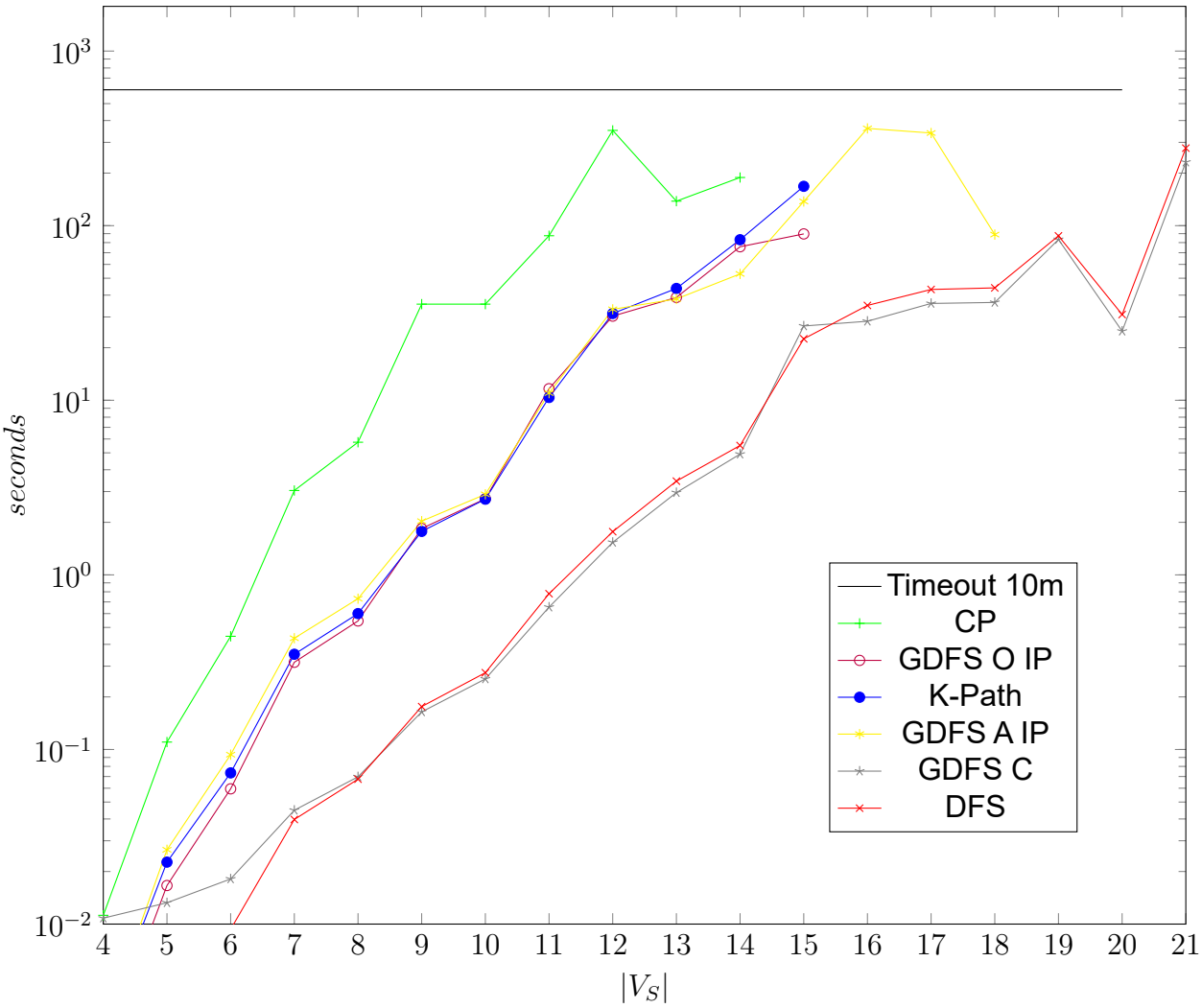


Figure 8.2: A comparison of the speed of path iterators on directed graphs without a vertex disjoint subgraph homeomorphism. The time taken is measured of exploring the entire search tree without pruning or contraction and "refuse longer paths" enabled. The source graph has average degree 2.429 and the random target graph has 50% more vertices and average degree 3.425. Neither graphs have labels

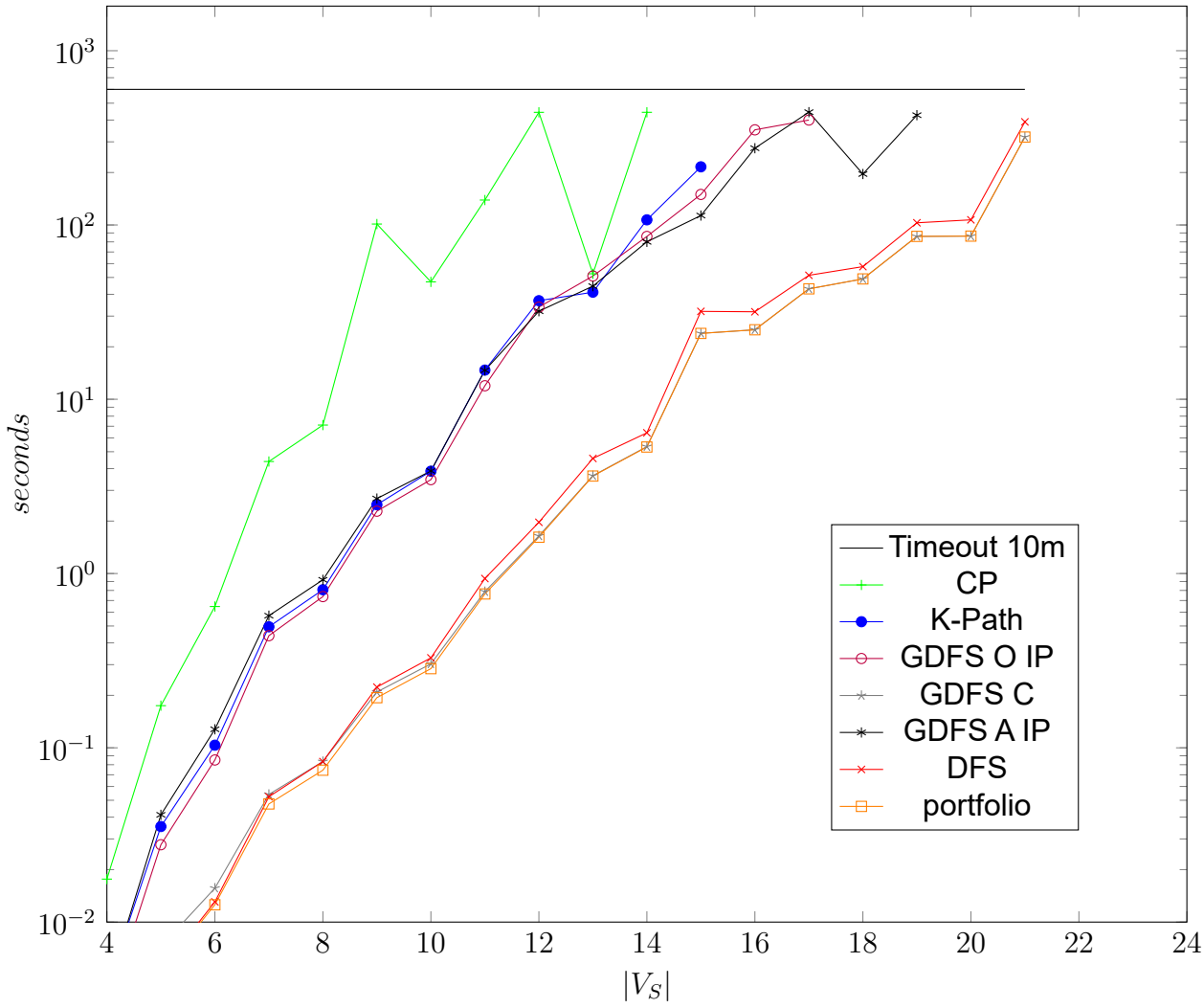


Figure 8.3: A comparison of the speed of path iterators on directed graphs without a vertex disjoint subgraph homeomorphism. The time taken is measured of exploring the entire search tree without pruning or contraction and “refuse longer paths” enabled. The source graph has average degree 2.429 and the random target graph has 50% more vertices and average degree 3.425. The source and target graphs have the same labels and distribution of them as the use case models

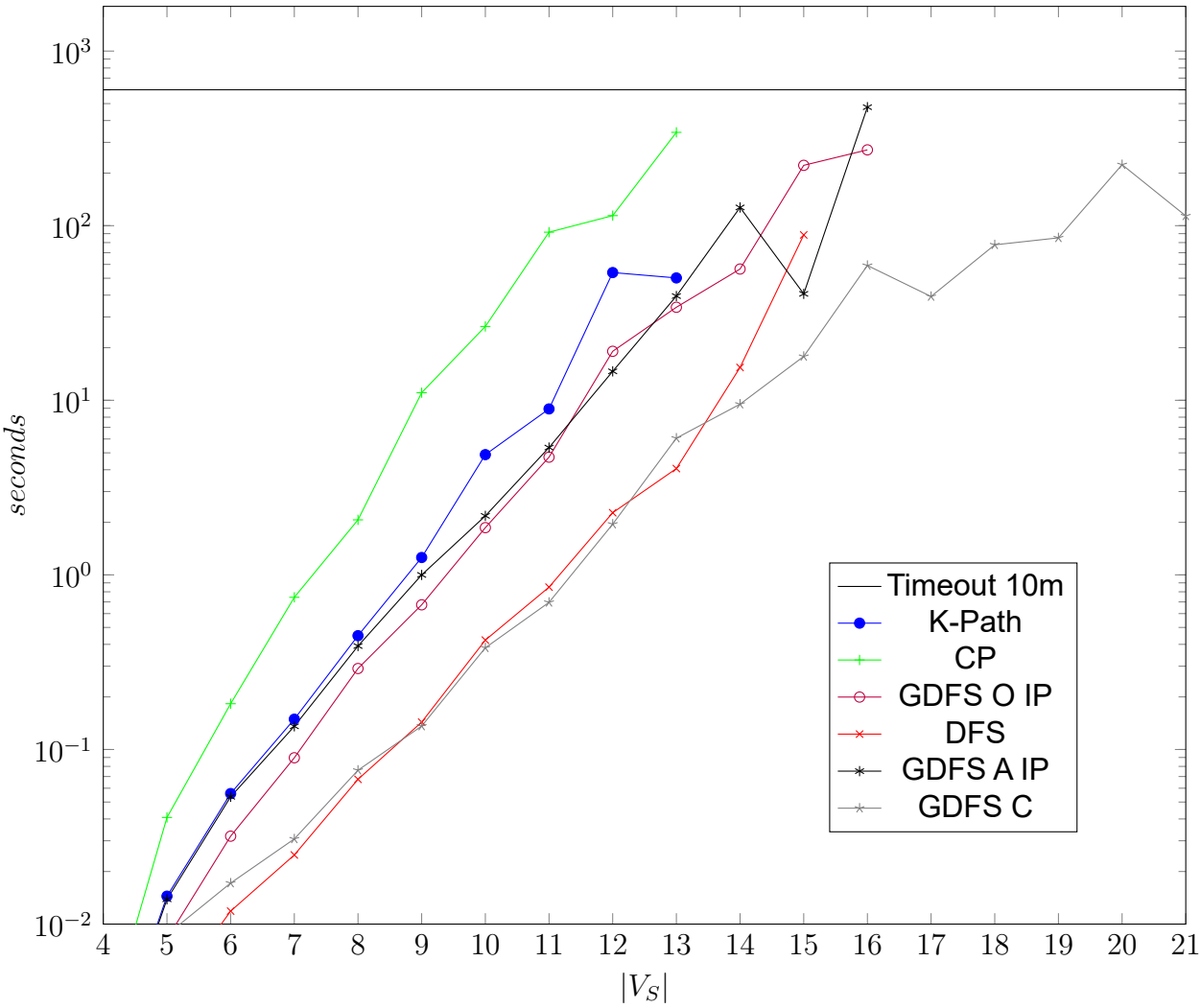


Figure 8.4: A comparison of the speed of path iterators on directed graphs with a vertex disjoint subgraph homeomorphism. The time taken is measured of exploring the entire search tree without pruning or contraction and "refuse longer paths" enabled. The source graph has average degree 3.0 and the random target graph has 50% more vertices and average degree 4.0.

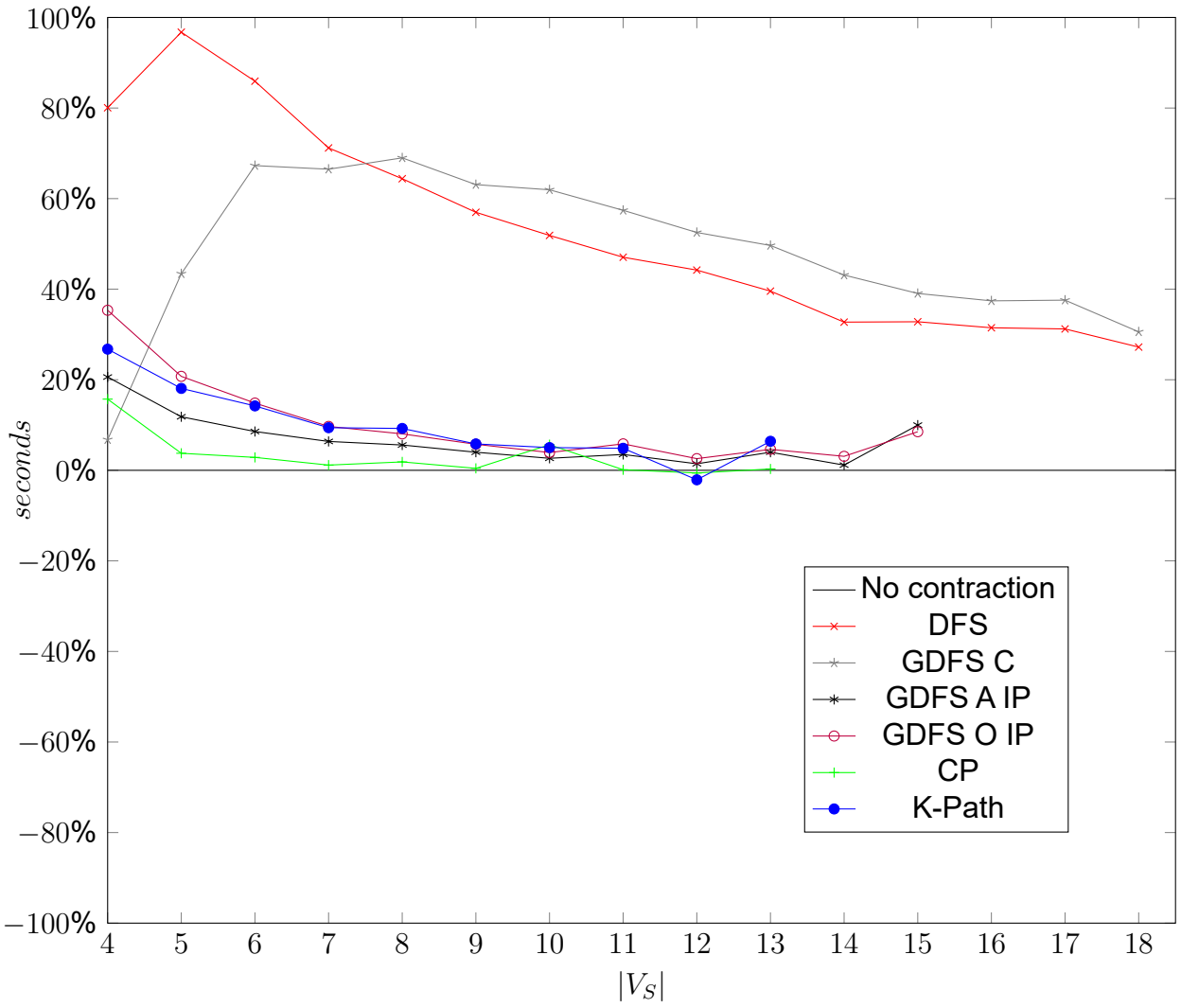


Figure 8.5: Relative time consumption of contraction-enabled subgraph homeomorphism search compared to contraction-disabled for different path iteration methods.

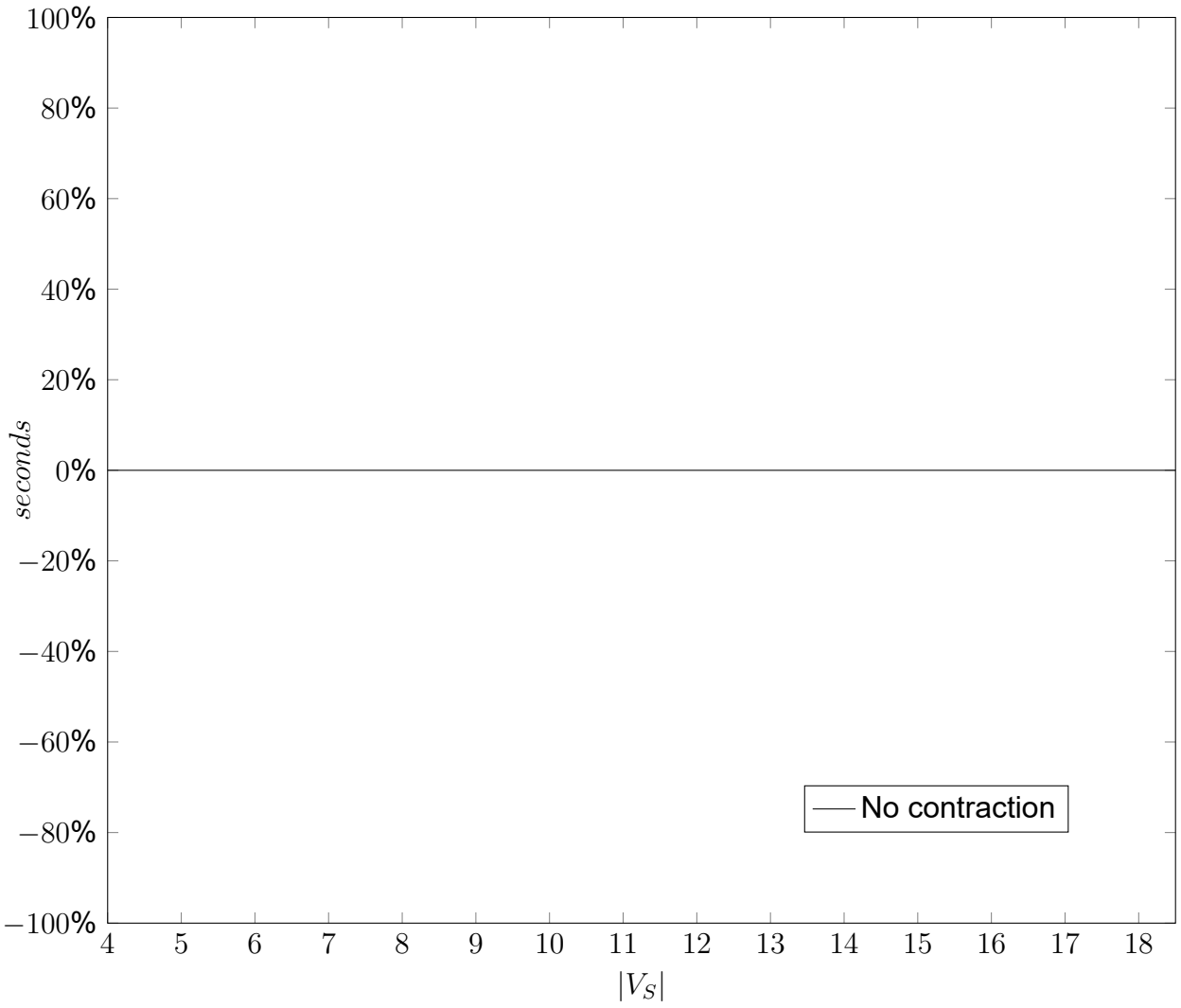


Figure 8.6: Relative time consumption of contraction-enabled subgraph homeomorphism search compared to contraction-disabled for different path iteration methods.

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

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

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

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

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

Table 8.4: The performance benefit (seconds) of contraction on random source graphs and random target graphs.

Pruning strategy	Filtering	Application	Effect (+)	Effect (-)
None			+0.0%	+0.0%
Zero domain	Label/degree	Serial		
		Cached		
		Parallel		
	Free neighbours	Serial		
		Cached		
		Parallel		
	M-reachability	Serial		
		Cached		
		Parallel		
	N-reachability	Serial		
		Cached		
		Parallel		
AllDifferent	Label/degree	Serial		
		Cached		
		Parallel		
	Free neighbours	Serial		
		Cached		
		Parallel		
	M-reachability	Serial		
		Cached		
		Parallel		
	N-reachability	Serial		
		Cached		
		Parallel		

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

Pruning strategy	Filtering	Application	Effect (+)	Effect (-)
None			+0.0%	+0.0%
Zero domain	Label/degree	Serial		
		Cached		
		Parallel		
	Free neighbours	Serial		
		Cached		
		Parallel		
	M-reachability	Serial		
		Cached		
		Parallel		
	N-reachability	Serial		
		Cached		
		Parallel		
AllDifferent	Label/degree	Serial		
		Cached		
		Parallel		
	Free neighbours	Serial		
		Cached		
		Parallel		
	M-reachability	Serial		
		Cached		
		Parallel		
	N-reachability	Serial		
		Cached		
		Parallel		

Table 8.6: The performance effect (seconds) of different pruning methods with- and without Connectivity check (CC) and neighbour check (NC) on random source graphs of size $|V| \in \{8..12\}$ with edge density 3.0 and a square of four ECP5 times as target graph. Effect (+) denotes cases with some homeomorphism and Effect (-) denotes cases without a homemorphism.

9 CONCLUSION

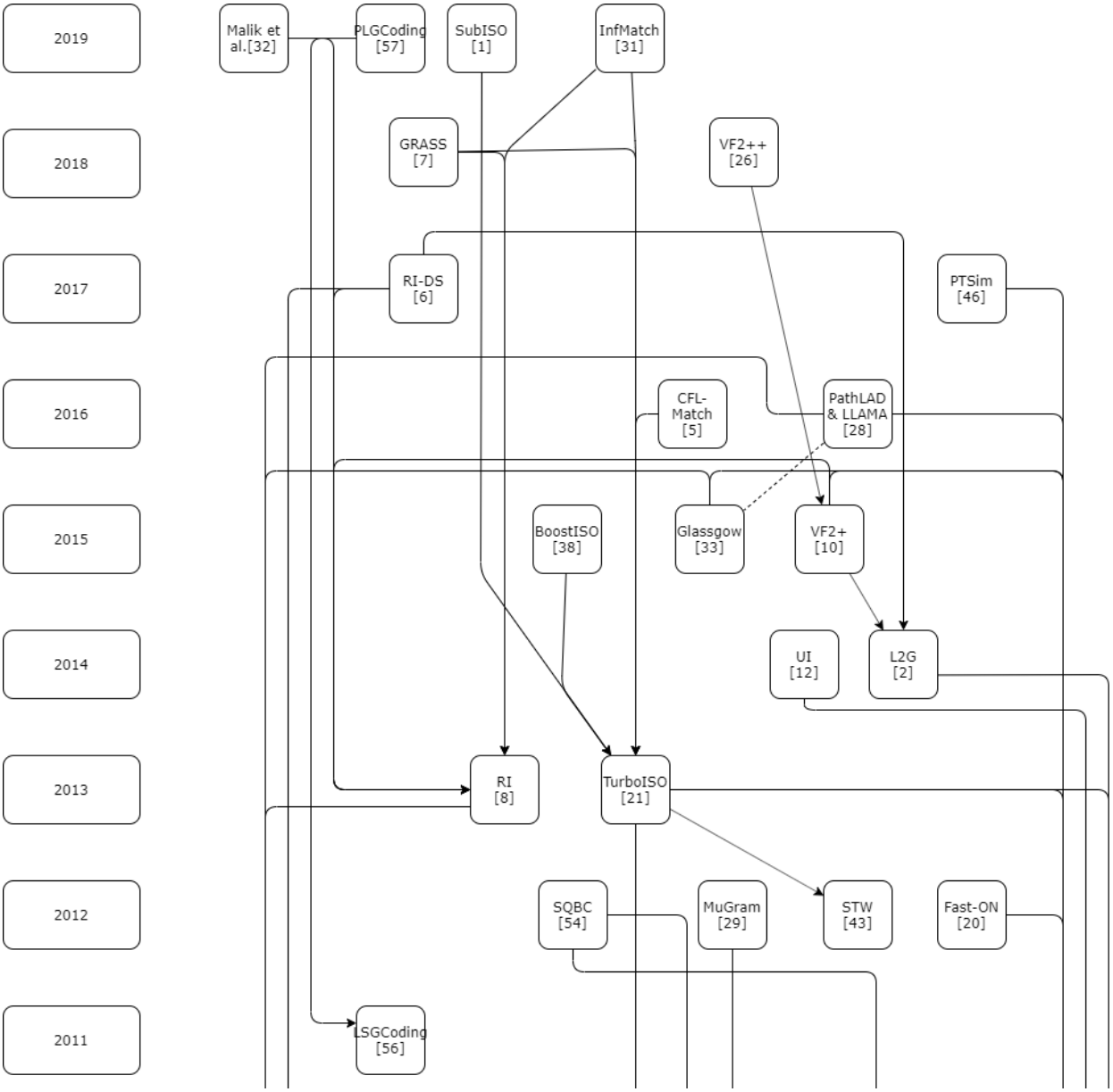
10 DISCUSSION

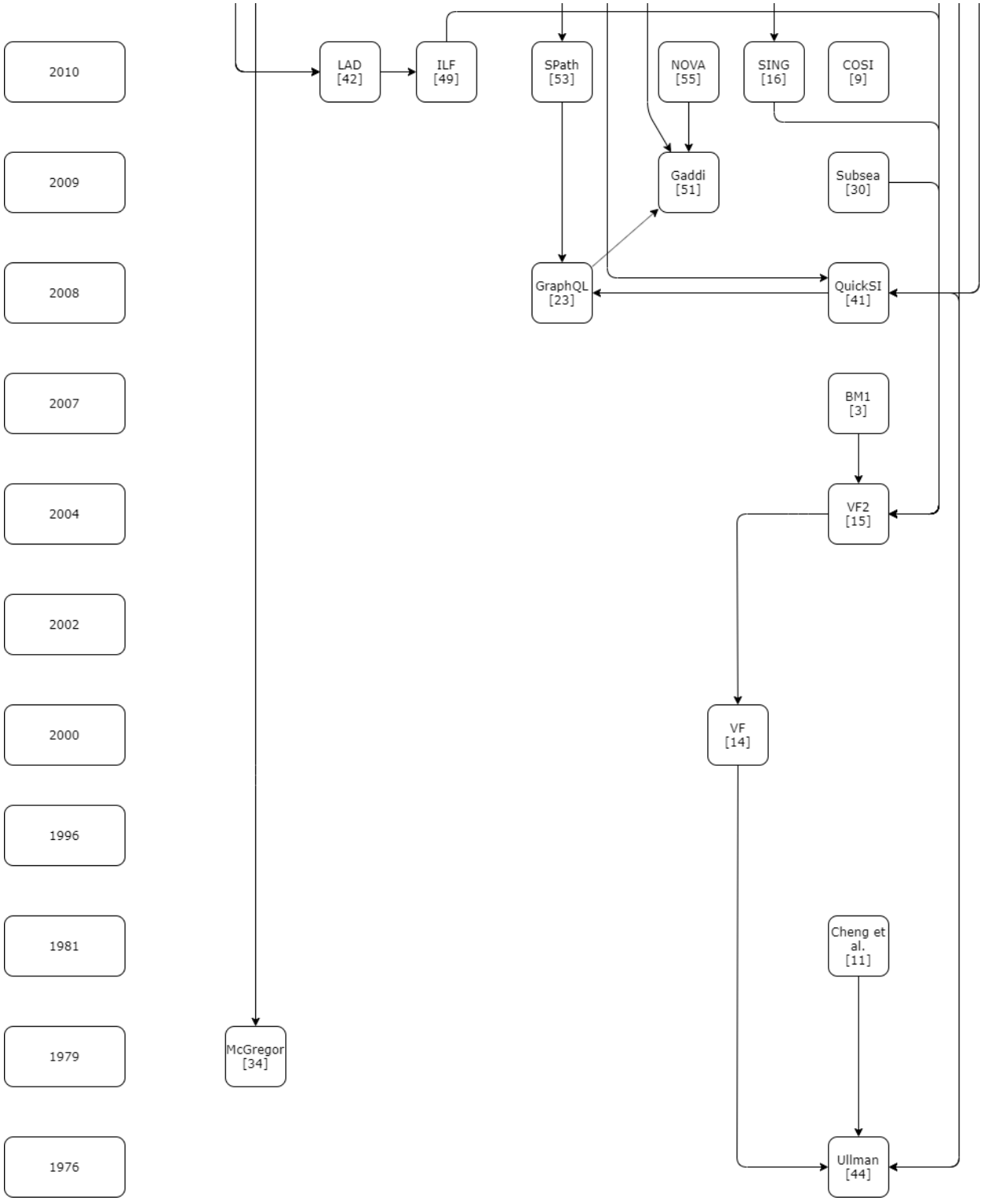
11 FUTURE RESEARCH

1. concurrent tree exploration
2. Backmarking and backjumping
3. choose target graph vertices based on some location heuristic, e.g. tiles
4. use hierachy: e.g. when a component has been matched with some pattern Q, immediately match the next component with an isomorphic pattern Q', allowing for backtracking
5. pruning does not take contracted vertices into account
6. multigraph introduces repetition

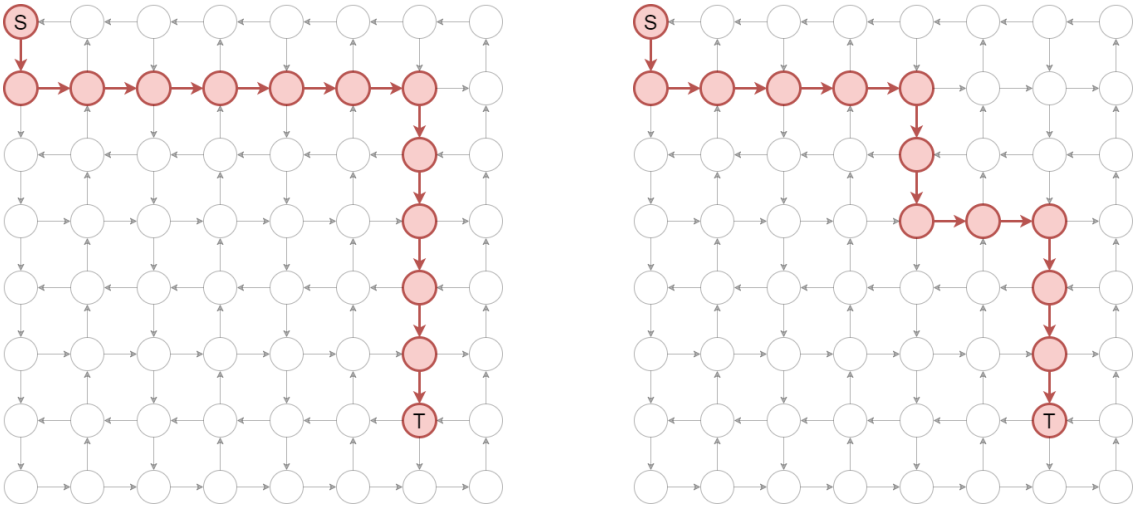
Appendices

.1 History of subgraph isomorphism algorithms



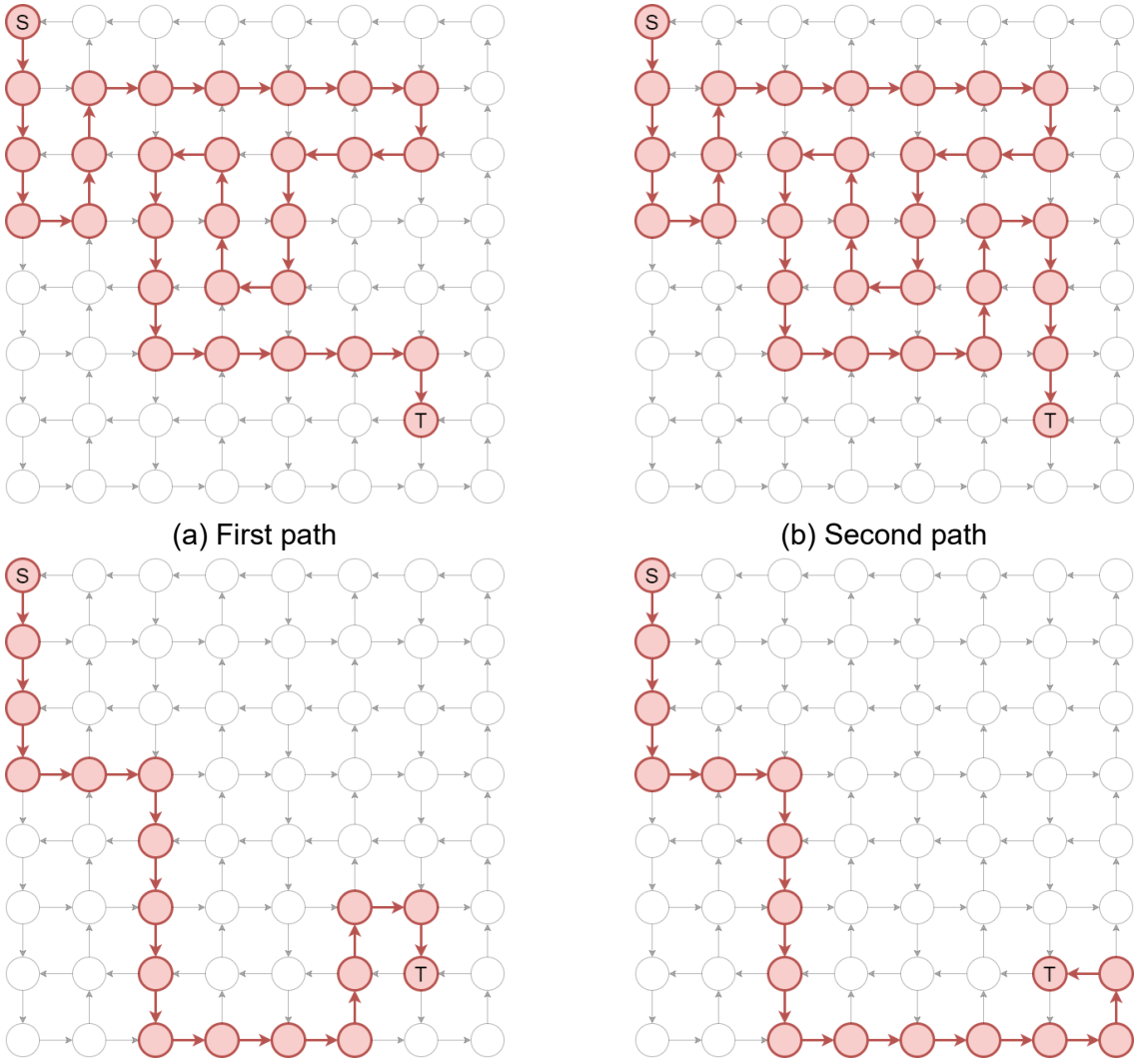

2010
2009
2008
2007
2004
2002
2000
1996
1981
1979
1976
LAD [42]
ILF [49]
SPath [53]
NOVA [55]
SING [16]
COSI [9]
Gaddi [51]
Subsea [30]
GraphQL [23]
QuickSI [41]
BM1 [3]
VF2 [15]
VF [14]
Cheng et al. [11]
McGregor [34]
Ullman [44]

.2 Examples of path iterators



(a) First path (b) Second path

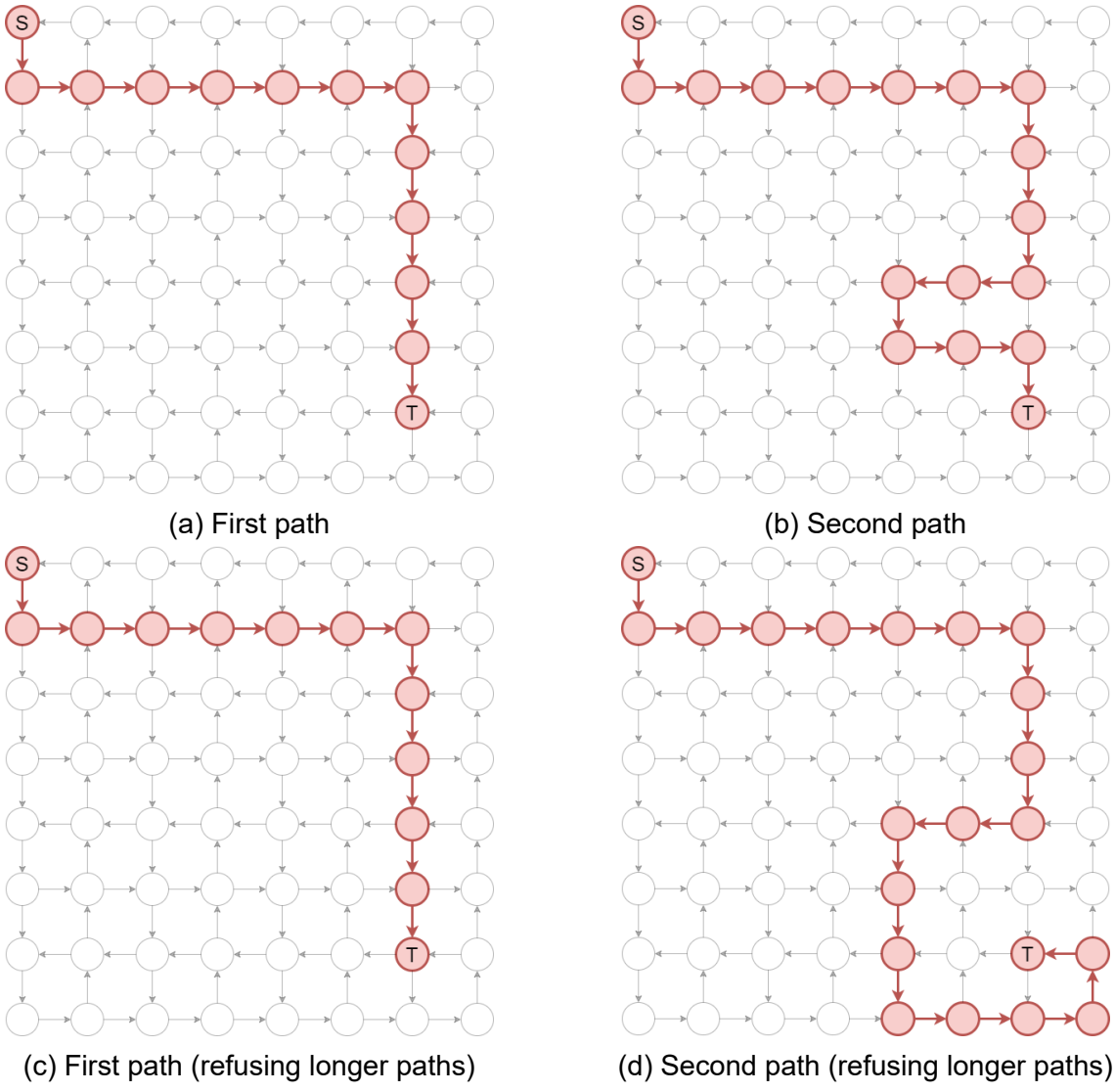
Figure 1: The first two iterations of the K-path path iterator in a square graph. This example is unaffected by “refusing longer paths”.



(a) First path (b) Second path

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

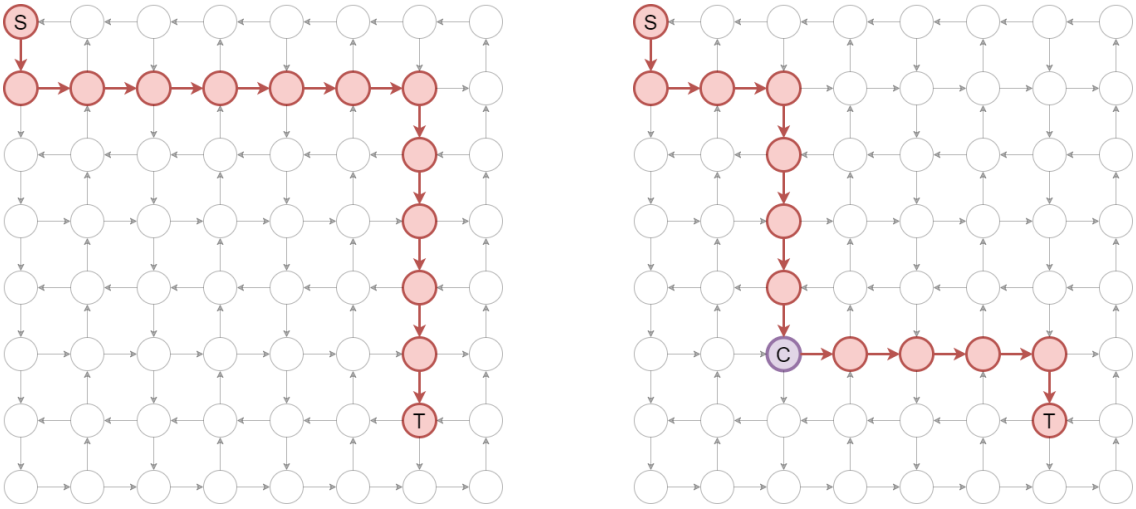
Figure 2: The first two iterations of the DFS path iterator in a square graph.



(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 verted disjoint subgraph homeomorphism

Proof. Let $M_V^{S'toS}$ and $M_E^{S'toS}$ be the (injective) vertex-to-vertex mapping and (injective) edge-to-path mapping from S_{cont} to S, respectively, and let (M_V^{StoT}, M_E^{StoT}) be some vertex disjoint subgraph homeomorphism from S to T, respectively.

We then construct a vertex-to-vertex mapping $M_V^{S'toT}$ by mapping each vertex $v \in V_{S'}$ to $M_V^{StoT}(M_V^{S'toS}(v))$. Recall from the definition of vertex disjoint subgraph homeomorphism that $\forall u \in V_S.L_S(u) \subseteq L_T(M_V^{StoT}(u))$. Since for every vertex $v \in V_{S'}$ we have $M_V^{S'toS}(v) \in V_s$ (injectivity), it also holds that $\forall u \in V_{S'}.L_S(M_V^{S'toS}(u)) \subseteq L_T(M_V^{StoT}(M_V^{S'toS}(u)))$. Moreover, since labels of remaining vertices are preserved through contraction, we have $\forall v \in V_{S'}.L_{S'}(v) = L_S(M_V^{S'toS}(v))$. Therefore, it holds that: $\forall u \in V_{S'}.L_{S'}(u) \subseteq L_T(M_V^{StoT}(M_V^{S'toS}(u)))$. 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 $E_V^{S'toT}$. For each edge $(u, v) \in E_{S'}$, we take the edges of the path $M_E^{S'toS}(u, v)$ and concatenate the edges of the corresponding paths in T to obtain a new path p'. We then add $((u, v), p')$ to $E_V^{S'toT}$. Recall from the definition of vertex disjoint subgraph homeomorphism that $\forall(u_1, u_2) \in E_S.\exists p \in values(M_E^{StoT}).first(p) = M_V^{StoT}(u_1) \wedge last(p) = M_V^{StoT}(u_2) \wedge ((u_1, u_2), p) \in M_E^{StoT}$.

Since $E_V^{S'toT}$ is non-empty, we have $\forall(u_1, u_2) \in E_{S'}.\exists p \in values(M_E^{S'toT})$. Moreover, by choosing p for each (u_1, u_2) as $M_E^{S'toT}(u_1, u_2)$, we have:

$\forall(u_1, u_2) \in E_{S'}.\exists p \in values(M_E^{S'toT}).first(p) = M_V^{S'toT}(u_1) \wedge last(p) = M_V^{S'toT}(u_2) \wedge ((u_1, u_2), p) \in M_E^{S'toT}$ which satisfies the second prerequisite.

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

$$\exists p \in \textbf{\textit{values}}(M_E^{StoT}).\exists x \in \textbf{\textit{intermediate}}(p).\exists p' \in (\textbf{\textit{values}}(M_E^{StoT}) \setminus \{p\}).x \in \textbf{\textit{intermediate}}(p')$$

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 $u \in V_S$ have indegree 1 and outdegree 1, qualifying it for contraction. Let the edges be $(\textbf{\textit{prec}}(u), u)$ and $(u, \textbf{\textit{succ}}(u))$ where $\textbf{\textit{prec}}(u)$ may be equal to $\textbf{\textit{succ}}(u)$. We know that $M_E^{StoT}(\textbf{\textit{prec}}(u), u)$ is internally vertex disjoint from all other paths in $\textbf{\textit{values}}(M_E^{StoT})$, as well as $M_E^{StoT}(u, \textbf{\textit{succ}}(u))$. They share at least an end vertex $M_E^{StoT}(u)$ and possibly $M_E^{StoT}(\textbf{\textit{prec}}(u))$ if $\textbf{\textit{prec}}(u) = \textbf{\textit{succ}}(u)$. We know that $M_E^{StoT}(u)$ is not the end vertex of another path since u 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 $\textbf{\textit{intermediate}}(M_E^{StoT}(\textbf{\textit{prec}}(u), u))$, $\textbf{\textit{intermediate}}(M_E^{StoT}(u, \textbf{\textit{succ}}(u)))$ and the vertex $M_V^{StoT}(u)$ that are not intermediate vertices of other paths in $values(M_E^{StoT})$.

When we contract this vertex, we get a new edge $(\textbf{\textit{prec}}(u), \textbf{\textit{succ}}(u))$ with start vertex $M_E^{StoT}(\textbf{\textit{prec}}(u))$, end vertex $M_E^{StoT}(\textbf{\textit{succ}}(u))$, and as intermediate vertices

intermediate$(M_E^{StoT}($*prec*$(u), u)) \cup$ *intermediate*$(M_E^{StoT}(u,$ *succ*$(u))) \cup M_V^{StoT}(u)$. The vertex set used in T remains exactly the same, thus does still not share intermediate vertices with other paths in $values(M_E^{StoT})$.

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. 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)

16. 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)
17. 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)
18. Donath, W.E.: Complexity Theory and Design Automation. In: 17th Design Automation Conference, pp. 412–419 (1980)
19. 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)
20. 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)
21. Gouda, K., and Hassaan, M.: A fast algorithm for subgraph search problem. In: DE53–DE58 (2012)
22. Grohe, M., Kawarabayashi, K.-I., Marx, D., and Wollan, P.: Finding topological subgraphs is fixed-parameter tractable. In: pp. 479–488 (2011)
23. 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)
24. Hauck, S., and DeHon, A.: Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2007)
25. 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)
26. 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)
27. Al-Hyari, A.: Towards Smart FPGA Placement Using Machine Learning (2019).
28. Ikram, J., and Mohsen, M.: FPGA Implementation of a Quantum Cryptography Algorithm. Smart Innovation, Systems and Technologies 146, 172–181 (2020)
29. Jüttner, A., and Madarasi, P.: VF2++—An improved subgraph isomorphism algorithm. Discrete Applied Mathematics 242, 69–81 (2018)
30. Kanazawa, K., and Maruyama, T.: An approach for solving large SAT problems on FPGA. ACM Transactions on Reconfigurable Technology and Systems 4(1) (2010)
31. Kotthoff, L., McCreesh, C., and Solnon, C.: Portfolios of subgraph isomorphism algorithms (2016)
32. 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)
33. 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)
34. Lingas, A., and Wahlen, M.: An exact algorithm for subgraph homeomorphism. Journal of Discrete Algorithms 7(4), 464–468 (2009)
35. 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)
36. 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)
37. 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)
38. Malík, J., Suchý, O., and Valla, T.: Efficient Implementation of Color Coding Algorithm for Subgraph Isomorphism Problem (2019)

39. McCreesh, C., and Prosser, P.: A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs (2015)
40. McGregor, J.: Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. Information Sciences 19(3), 229–250 (1979)
41. Miyamoto, M., Iwamura, M., Kise, K., and Gall, F.L.: Quantum Speedup for the Minimum Steiner Tree Problem. In: Kim, D., Uma, R.N., Cai, Z., and Lee, D.H. (eds.) Computing and Combinatorics, pp. 234–245. Springer International Publishing, Cham (2020)
42. Nawari, M., Ahmed, H., Hamid, A., and Elkhidir, M.: FPGA based implementation of elliptic curve cryptography. In: Institute of Electrical and Electronics Engineers Inc. (2015)
43. Régin, J.-C.: Filtering algorithm for constraints of difference in CSPs. In: pp. 362–367. AAAI, Menlo Park, CA, United States (1994)
44. Ren, X., and Wang, J.: Exploiting Vertex Relationships in Speeding up Subgraph Isomorphism over Large Graphs. Proc. VLDB Endow. 8(5), 617–628 (2015)
45. Rudolf, M.: Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching. In: Ehrig, H., Engels, G., Kreowski, H.-J., and Rozenberg, G. (eds.) Theory and Application of Graph Transformations, pp. 238–251. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
46. Saha, S., Alam, M., and Mondol, R.: FPGA implementation of FlexRay protocol with built-in-self-test capability. In: Institute of Electrical and Electronics Engineers Inc. (2014)
47. Shang, H., Zhang, Y., Lin, X., and Yu, J.X.: Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. Proc. VLDB Endow. 1(1), 364–375 (2008)
48. Solnon, C.: AllDifferent-based filtering for subgraph isomorphism. Artificial Intelligence 174(12-13), 850–864 (2010)
49. Sun, Z., Wang, H., Wang, H., Shao, B., and Li, J.: Efficient subgraph matching on billion node graphs. In: pp. 788–799. Association for Computing Machinery (2012)
50. Ullmann, J.R.: An Algorithm for Subgraph Isomorphism. J. ACM 23(1), 31–42 (1976)
51. Wolf, C.: Yosys Open SYnthesis Suite. (2016)
52. Xiao, Y., Wu, W., Wang, W., and He, Z.: Efficient Algorithms for Node Disjoint Subgraph Homeomorphism Determination. In: Haritsa, J.R., Kotagiri, R., and Pudi, V. (eds.) Database Systems for Advanced Applications, pp. 452–460. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
53. Xie, X., Li, Z., and Zhang, H.: Efficient Subgraph Matching in Large Graph with Partitioning Scheme. In: Proceedings - 13th Web Information Systems and Applications Conference, WISA 2016 - In conjunction with 1st Symposium on Big Data Processing and Analysis, BDPA 2016 and 1st Workshop on Information System Security, ISS 2016, pp. 28–33 (2017)
54. Yalla, P., and Kaps, J.-P.: Lightweight cryptography for FPGAs. In: pp. 225–230 (2009)
55. Yen, J.Y.: Finding the K Shortest Loopless Paths in a Network. Management Science 17(11), 712–716 (1971)
56. Yu, H., Lee, H., Lee, S., Kim, Y., and Lee, H.-M.: Recent advances in FPGA reverse engineering. Electronics (Switzerland) 7(10) (2018)
57. Zampelli, S., Deville, Y., and Solnon, C.: Solving subgraph isomorphism problems with constraint programming. Constraints 15(3), 327–353 (2010)
58. Zhang, S., Hu, M., and Yang, J.: TreePi: A novel graph indexing method. In: Proceedings - International Conference on Data Engineering, pp. 966–975 (2007)
59. 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)
60. 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)
61. Zhao, P., and Han, J.: On Graph Query Optimization in Large Networks. Proc. VLDB Endow. 3(1–2), 340–351 (2010)

62. 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)
63. 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)
64. Zhu, L., and Song, Q.: A study of Laplacian spectra of graph for subgraph queries. In: pp. 1272–1277 (2011)
65. 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)