Scalable Pattern Matching in Computation Graphs

Luca Mondada
University of Oxford
Oxford, UK
Quantinuum Ltd
Cambridge, UK
luca.mondada@cs.ox.ac.uk

Pablo Andrés-Martínez
Quantinuum Ltd
Cambridge, UK

Graph rewriting is a popular tool for the optimisation and modification of graph expressions in domains such as compilers, machine learning and quantum computing. The underlying data structures are often port graphs—graphs with labels at edge endpoints. A pre-requisite for graph rewriting is the ability to find graph patterns. We propose a new solution to pattern matching in port graphs. Its novelty lies in the use of a pre-computed data structure that makes the pattern matching runtime complexity independent of the number of patterns. This offers a significant advantage over existing solutions for use cases with large sets of small patterns.

Our approach is particularly well-suited for quantum superoptimisation. We provide an implementation and benchmarks showing that our algorithm offers a 20x speedup over current implementations on a dataset of 10 000 real world patterns describing quantum circuits.

1 Introduction

Optimisation of computation graphs is a long-standing problem in computer science that is seeing renewed interest in the compiler [12], machine learning (ML) [8, 5] and quantum computing communities [20, 19]. In all of these domains, graphs encode computations that are either expensive to execute or that are evaluated repeatedly over many iterations, making graph optimisation a primary concern.

Domain-specific heuristics are the most common approach in compiler optimisations [14, 17]— a more flexible alternative are optimisation engines based on *rewrite rules*, describing the allowable graph transformations [3, 4]. Given a computation graph as input, we find a sequence of rewrite rules that transform the input into a computation graph with minimal cost. One successful approach in both ML and quantum computing has been to use automatically generated rules, scaling to using hundreds and even thousands of rules [20, 8, 19].

In the implementations cited above, pattern matching is carried out separately for each pattern, becoming a bottleneck for large rule sets. We present an algorithm for pattern matching on computation graphs that uses a pre-computed data structure to return all pattern matches in a single query. The set of rewrite rules are directly encoded in this data structure: after a one-time cost for construction, pattern matching queries can be answered in running time independent of the number of rewrite rules.

We provide a novel solution to pattern matching on port graphs [6] with a runtime complexity independent of the number of patterns. As a trade-off, the runtime may be exponential in the size of the patterns. For pattern sizes of practical interest in quantum computing, however, the resulting costs are manageable: the exponential scaling is in the number of qubits of the patterns, which is bounded by a single digit constant in relevant rewriting use cases [20].

The solution we propose can be seen as an adaptation of Rete networks [7] to the special case of port graph pattern matching. The additional structure obtained from restricting our considerations to graphs results in a simplified network design and crucially, allows us to derive worst-case asymptotic

Submitted to:
GCM 2024

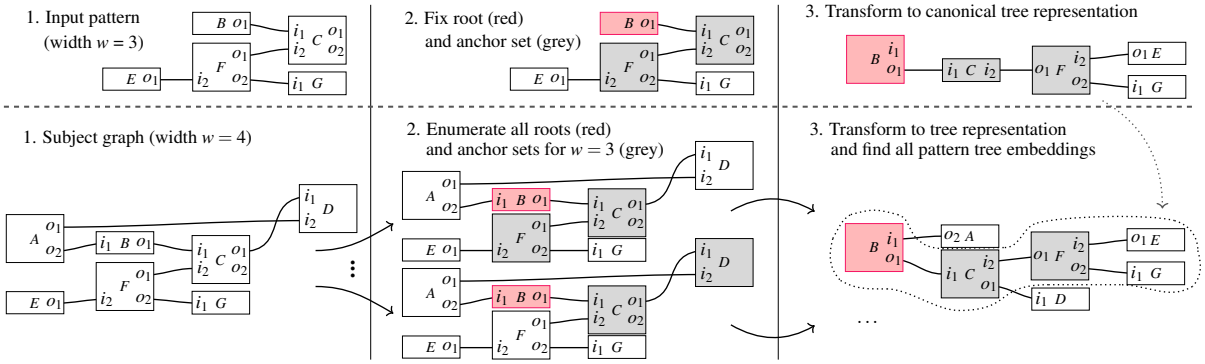


Figure 1: Pattern matching on a port graph is reduced to the problem of matching on trees. A subset of vertices are chosen as anchor sets (left). A neighbourhood of the anchors is extracted and represented as a tree (middle). Finally, pattern matches are found by searching for matching subtrees (right).

runtime bounds—overcoming a key limitation of Rete. A similar problem is also studied in the context of multiple-query optimisation for database queries [16, 15], but has limited itself to developing caching strategies and search heuristics for specific use cases. Finally, using a pre-compiled data structure for pattern matching was already proposed in [13]. However, with a $n^{\Theta(m)}$ space complexity—n is the input size and m the pattern size—it is a poor candidate for pattern matching on large graphs, even for small patterns.

2 Paper overview

Taking advantage of *port labels* on graph data structures leads to a speedup for pattern matching over the general case [10]. Port labels are data assigned to every endpoint of the edges of a graph, such that the labels at every vertex are unique. Such labels can for instance be assigned to processes with distinguishable inputs and outputs: a function that maps inputs $(x_1, \ldots, x_n)$ to output $(y_1, \ldots, y_m)$ can assign labels i_1 to i_n and o_1 to o_m to its incident edges in the computation graph. The resulting data structure is a *port graph* [6].

Main idea. For a set of ℓ pattern port graphs $P_1, \ldots, P_\ell$ and a subject port graph G, we solve the problem of finding all pattern embeddings $P_i \to G$. We distinguish two separate stages during pattern matching:

> *Pre-computation stage.* Compile the set of patterns into a data structure designed to speed up later queries. In the process, we select for every pattern P_i an *anchor set*, i.e. a subset X_i of vertices in P_i. The input port graph G is not required at this stage and, hence, this computation is done only once for a given collection of patterns $P_1, \ldots, P_\ell$.
>
> *Fast pattern matching stage.* Given G, compute for each P_i all embeddings $P_i \to G$. This is achieved by enumerating all possible images X in G of pattern anchor sets $X_1, \ldots, X_\ell$ and finding the subset of patterns i for which the map $X_i \to X$ can be extended to a valid pattern embedding of P_i in G.

The pattern matching stage can be decomposed into known problems by showing that embeddings with fixed anchor sets can be equivalently seen as rooted tree embeddings. The enumeration of valid choices of X in G then becomes a tree counting argument. Meanwhile, the set of patterns that embed in G for a fixed $X_i \to X$ is obtained from a pre-computed decision tree. An overview is presented in fig. 1.

Results and contributions. Our first major contribution is a pattern matching algorithm for port graphs with a runtime complexity bound independent of the number of patterns being matched, achieved using a one-off pre-computation. The main complexity result is expressed in terms of maximal pattern *width* w and *depth* d, two measures of pattern size defined in section 3.1. These are directly related to the tree representation illustrated in fig. 1: width is equal to the size of the anchor set (proposition 4) and depth is at most twice the tree height (eq. (7)). We assume bounded degree graphs (the complete list of assumptions is given in section 3.2) and we use the *graph size* $|G|$ to refer to the number of vertices in G.

Theorem 1. *Let $P_1, \dots, P_\ell$ be patterns with at most width w and depth d. The pre-computation runs in time and space complexity*

$$O\left((d \cdot \ell)^w \cdot \ell + \ell \cdot w^2 \cdot d\right).$$

For any subject port graph G, the pre-computed prefix tree can be used to find all pattern embeddings $P_i \to G$ in time

$$O\left(|G| \cdot \frac{c^w}{w^{1/2}} \cdot d\right) \tag{1}$$

where $c = 6.75$ is a constant.

The runtime complexity is dominated by an exponential scaling in maximal pattern width w. Meanwhile, the advantage of our approach over matching one pattern at a time grows with the number of patterns ℓ. It is thus of particular interest for matching numerous small width patterns.

We illustrate this point by comparing our approach to a standard algorithm that matches one pattern at a time [10], with runtime complexity $O(\ell \cdot |P| \cdot |G|)$. Using $|P| \leq w \cdot d$ (shown in section 3.1) and comparing to eq. (1), we thus have a speedup in the regime $\Theta(c^w/w^{3/2}) < \ell$. On the other hand, ℓ is upper bounded by the maximum number $N_{w,d}$ of patterns of bounded width and depth. Using a crude lower-bound for $N_{w,d}$ derived in appendix B, we obtain a computational advantage for our approach when

$$\Theta\left(\frac{c^w}{w^{3/2}}\right) < \ell < \left(\frac{w}{2e}\right)^{\Theta(wd)} \leq N_{w,d}. \tag{2}$$

Our second major contribution is an efficient Rust library for port graph pattern matching[1]. We present benchmarks on a real world dataset of 10 000 quantum circuits in section 5, showing a 20× speedup over a leading C++ implementation of pattern matching for quantum circuits.

3 Preliminaries

3.1 Definitions

A port graph G is a tuple $G := (V, E, \mathscr{P}, \lambda)$ where (V, E) is an undirected multigraph, $\mathscr{P}$ is a finite set of *port labels* and $\lambda : V \times \mathscr{P} \rightharpoonup E \cup \{\omega\}$ is a partial function that on its domain of definition either assigns port labels to edges or marks them as open ports using a specially dedicated symbol ω. The port graph is valid if $\lambda(v,p) = \lambda(v,p') = e \in E$ if and only if e is an edge incident in v and $p = p'$. We then say that e is attached to v at port p. The domain of definition of $\lambda(v,\cdot)$ is the set of port labels at vertex v, written $ports(v)$. The degree of v is $deg(v) = |ports(v)|$. It will often be convenient to leave the definition of λ implicit and for an edge $e = \lambda(v,p) = \lambda(v',p')$, to write it instead as the set $e = \{(v,p),(v',p')\}$. Additionally, we may consider port graphs with labelled vertices, determined by *vertex label maps* $V \to \mathscr{W}$, where $\mathscr{W}$ is a set of labels.

[1] portmatching: `https://github.com/lmondada/portmatching`

Width, depth and linear paths. Fix a partition of port labels $\mathscr{P}$ into pairs of elements (and an additional singleton set if $|\mathscr{P}|$ is odd). This defines an equivalence relation $\sim$ where $p \sim p'$ if p and p' are in the same partition. The relation $\sim$ defines a partition of the edges of a port graph into paths. A linear path in a port graph G with edges E and port labels in $\mathscr{P}$ is a path $P \subseteq E^*$ such that for every vertex v in G and ports $p, p' \in \mathscr{P}$ satisfying $p \sim p'$,

$$\lambda(v,p) \in P \quad \Longrightarrow \quad \lambda(v,p') \in P \cup \{\omega\}. \tag{3}$$

From a single edge in G, a linear path can be constructed uniquely by repeatedly appending edges to the path so that (3) is satisfied. The linear path decomposition of G is the unique partition of the edges of G into linear paths. The width $width(G)$ of G is the number of linear paths in this decomposition. The depth $depth(G)$ of G is the length of the longest linear path in G. Every edge is on exactly one linear path, while vertices may be on zero, one or several linear paths. We have always $|G| \leq width(G) \cdot depth(G)$.

Patterns and embeddings. A pattern is a connected port graph. A pattern embedding (or just embedding) $\varphi : P \to G$ from a pattern $P = (V_P, E_P, \mathscr{P}, \lambda_P)$ to a subject port graph $G = (V, E, \mathscr{P}, \lambda)$, both with identical port label sets, is given by an injective vertex map $\varphi_V : V_P \to V$ such that $\lambda_P(v,p)$ is defined if and only if $\lambda(\varphi_V(v), p)$ is defined and the edge map $\varphi_E : E_P \to E$ defined as

$$\varphi_E(e) = \lambda(\varphi_V(v), p) \quad \text{for } (v,p) \in V \times \mathscr{P} \text{ s.t. } \lambda_P(v,p) = e \tag{4}$$

is well-defined and injective. Finally, if the pattern and port graphs have node labels $V_P \to \mathscr{W}$ and $V \to \mathscr{W}$, then we also require that pattern embeddings preserve those.

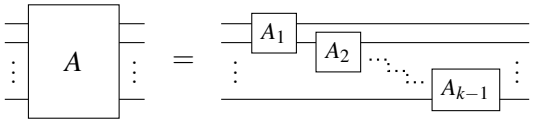
3.2 Simplifying assumptions

We list here a series of assumptions made throughout our argument. They represent a restriction from the most general case but we do not find that they restrict the usefulness of the result in practice. We show in section 3.3 that these assumptions hold in the case of quantum circuits. Moreover, as discussed in section 5, none of these assumptions are required for the implementation, and we have not observed any impact on performance when lifting them in practice, so we conjecture that these assumptions can be loosened and our results generalised.

1. All graphs and patterns are of bounded maximum degree Δ.
2. There is no linear path that forms a cycle.
3. All pattern embeddings $\varphi : P \to G$ must be *convex*, i.e. for every subgraph $H \subseteq G$ that contains the image of P, $\varphi(P) \subseteq H$, it holds that $width(P) \leq width(H)$.

Note also that eq. (4) requires that the degree of a vertex v in P is also preserved $deg(v) = deg(\varphi(v))$. Importantly, a pattern embedding may map a vertex with open port p to a vertex in the subject graph that has an edge attached to port p.

We will further simplify the problem by making choices of presentation that do not imply any loss of generality. First of all, we will assume that all vertices are on at most two linear paths (and thus in particular $\Delta = 4$). Vertices on $k > 2$ linear paths can always be broken up into a composition of $k-1$ vertices, each on two linear paths as follows:



This transformation leaves graph width unchanged but may multiply the graph depth by up to a factor Δ. We can then fix the set of port labels to the set $\mathscr{P} = \{i_1, i_2, o_1, o_2\}$ with a total order $\leq$ on $\mathscr{P}$[2]. We fix the partition of $\mathscr{P}$ into pairs of elements given by $i_k \sim o_k$ for $k \in \{1,2\}$. We also enforce that at every vertex v, the set of ports $ports(v)$ is partitioned by $\sim$ into $\lfloor ports(v)/2 \rfloor$ pairs of elements and at most one singleton set. This can always be achieved by relabelling vertex ports. Finally, we assume that all patterns have the same width w and depth d, are connected port graphs and have at least 2 vertices.

Using these assumptions we can obtain the following notable bound on graph width.

Proposition 2. *Let G be a port graph with n_{odd} vertices of odd degree and n_ω open ports. Then the graph width of G is at most $\lfloor (n_{odd} + n_\omega)/2 \rfloor$.*

The proof is in the appendix A.1.

Rooted paths ordering. The total order on $\mathscr{P}$ also induces a total order on the paths $e_1 \cdots e_k \in E^*$ in G that start in the same vertex $r \in e_1$: the paths are equivalently described by a string in $\mathscr{P}^*$, the sequence of ports of $e_1, \ldots, e_k$, which we order using the lexicographical ordering on strings. For every vertex v in G there is thus a unique smallest path from r to v in G that is invariant under isomoprhism of the underlying graph (i.e. relabelling of the vertex set).

3.3 Quantum Circuits

We see pattern matching for quantum circuits as one of the main applications of our results. We therefore choose to introduce here the quantum circuit syntax as a motivation and illustration of the port graph formalism. Similar data structures are also in use in other parts of compilation science, variously referred to as circuits, computation graphs or dataflow graphs. Readers familiar with port graphs or uninterested in the application in quantum computing may skip directly to the definitions of section 3.1.

The set of operations in a quantum circuit is called the *gate set* of the computation and forms the set of node weights of the port graph. To every element of the gate set, called a *gate type*, is associated a gate arity. A gate with gate type of arity n has n incoming port labels i_k and n outgoing port labels o_k—by our assumptions we thus assume $1 \leq n \leq 2$. Edges always connect outgoing to incoming labels, and the directed port graph that results from these edge orientations is acyclic. A quantum circuit has q qubits if it has q outgoing and q incoming open ports: the inputs and outputs of the circuit.

All assumptions made in section 3.2 can be easily verified for quantum circuits and in fact will also apply to most computation graphs more generally. Indeed:

1. Bounded degree is a direct consequence of a having a fixed set of gate types.
2. For any directed acyclic computation, using port labels in i_k for incoming ports and o_k for outgoing labels will always result in non-cyclic linear paths.
3. Similarly, convexity is a natural restriction in the context of rewriting systems for acyclic digraphs, as it ensures that the application of a rewrite rule does not introduce a cycle in the graph.

Using the $i_k \sim o_k$ port label partition, linear paths in a quantum circuit correspond to the paths of gates along a single qubit. For applications to quantum circuits, we can thus bound graph width and depth as follows:

Proposition 3. *The port graph G of a quantum circuit with q qubit and at most d gates on any one qubit has width q and depth d.* □

Some further considerations on applying our work to quantum circuits are discussed in appendix C.

[2] Any total order will work, e.g. $i_1 \leq i_2 \leq o_1 \leq o_2$.

3.4 Pattern Matching

In our pattern matching task, given a subject port graph G and a collection of port graphs $P_1, \ldots, P_\ell$, we must find all pattern embeddings

$$\{\varphi : P_i \to G \mid 1 \leq i \leq \ell \text{ s.t. } \varphi \text{ is a pattern embedding}\}. \tag{5}$$

Finding pattern embeddings in port graphs is a simple problem already studied in other contexts [9, 10]. As a result of eq. (4), for every vertex r in P and r_G in G there can be at most one embedding $P \to G$ that maps r to r_G: for $v \neq r$ in P, there is a path in P from r to v which, viewed as a sequence of port labels, maps uniquely to a path in G starting at r_G and ending in the image of v. For a choice of r in P it is therefore sufficient to consider every posible image r_G in G to find all embeddings $P \to G$.

We are however interested in the regime where the number of patterns ℓ may be large and pattern matching is performed many times for the same set of patterns. For this scenario it makes sense to proceed in two steps and introduce *pattern matching with pre-computation*. Given patterns $P_1, \ldots P_\ell$, we first produce a *pattern matcher*, a program whose representation can be stored to disk. In a second step, a subject port graph G is passed to the pattern matcher, which computes the set (5). We are interested in two properties of the solution:

- What is the complexity of pattern matching on input G given such a pattern matcher?
- What is the time complexity of generating a pattern matcher given patterns, and what is the size of the pattern matcher that is produced?

The answer to the first question is our main concern: for a fixed collection of patterns, this will determine the runtime to obtain all pattern embeddings given an input diagram. The second question, on the other hand, primarily concerns a one-off pre-computation step that only needs to be performed once for any set of patterns. In practice, this may also impinge on the first question, if the matcher does not fit into RAM and/or CPU cache.

4 Algorithm description

4.1 Canonical Tree Representation

Connected port graphs admit an equivalent representation as trees, which we will use for matching.

Split Graphs. Let G be a connected port graph with vertices V and consider the linear path decomposition of G. In this decomposition every vertex v in G must be on one or more linear paths. We mark a subset $X \subseteq V$ of vertices of G as 'immutable' and split every other vertex $v \in V \setminus X$ into multiple vertices, rewiring the edges in such a way that all vertices not in X are now on exactly one linear path. We call the graph thus obtained the X-split graph of G, and write it $split_X(G)$. Figure 2 shows an example of a graph and its split graph. Formally, the split graph can be characterised using an equivalence relation $\equiv$ given by

$$(v, p) \equiv (v', p') \quad \Leftrightarrow \quad v = v' \wedge (v \in X \vee p \sim p') \tag{6}$$

The vertices of the split graph are the equivalence classes of $\equiv$ and the edges are obtained by mapping the edges of G one to one: two vertices in the X-split graph are connected by an edge if and only if there is an edge in G between some elements of their equivalence classes. Note that if the anchor set X is too small, $split_X(G)$ may not be connected; e.g. if $X = \varnothing$ there would be $width(G)$ connected components, one per linear path.

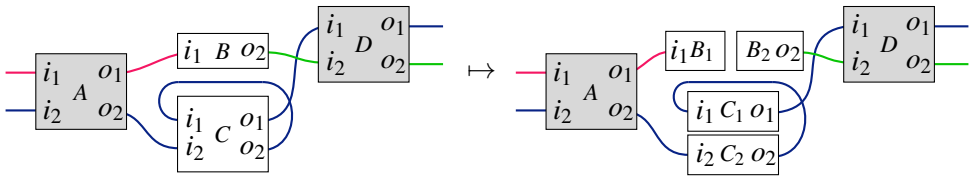


Figure 2: A port graph and its linear path decomposition (coloured edges) on the left. On the right, the split graph resulting from the choice of anchors $X = \{A, D\}$. We use the circuits convention, i.e. port labels are partitioned into linear paths using the relation $i_k \sim o_k$.

A recovery of the original port graph G given $\text{split}_X(G)$ is made possible by adding vertex labels to the split graph that identify split vertices. In the following, split graphs are always rooted trees, i.e. connected acyclic graphs with a chosen root vertex. Using paths of port labels we obtain a vertex labelling that is invariant under pattern embedding. This is discussed in more details in the context of the CT representation in appendix D.

Anchor sets. If $split_X(G)$ is connected and acyclic, we say that X is an anchor set of G and call the vertices in X anchor vertices. If $width(G) > 1$, then every linear path in G must contain at least one anchor vertex. A set of $width(G)$ anchors always exists and can be computed constructively:

Proposition 4. *For a connected port graph G of width w and depth d and for a vertex r of G, listing 1 returns an anchor set of w vertices; we call this the set of canonical anchors. Its runtime is $O(w^2 \cdot d)$.*

The proof is in appendix A.2. Note that the code given assumes that the linear paths of G are already computed. These can be computed at the beginning of the computation in time linear in the graph size—and thus do not affect the overall asymptotic complexity. As a direct consequence we have the following:

Corollary 5. *For a port graph G and root r_G in G: $width(G) = |\textsc{CanonicalAnchors}(G, r_G)|$.* □

CT representation. We call the tree $\text{split}_X(G)$ obtained from the canonical anchors, along with the choice of root r, the **canonical tree** (CT) representation of G. For simplicity, we will assume that the root is chosen such that it is on a single linear path[3]. Non-root internal nodes with more than one child are contained in the set of anchor vertices of G, every leaf is at most a height $depth(G)$ below the nearest anchor and there is at least one leaf a distance $depth(G)$ from the nearest anchor. As a consequence the CT tree width t_w and height t_h can be bound by

$$t_w \leq (\Delta - 2) \cdot width(G) = 2 \cdot width(G) \quad \text{and} \quad depth(G)/2 \leq t_h \leq width(G) \cdot depth(G), \tag{7}$$

where $\Delta = 4$ is the maximum degree of G. Using CT representations of patterns simplifies the pattern matching problem for three reasons:

- Connected port graphs are uniquely characterised by their CT representation, i.e. there is an injective map from patterns with a choice of root to their CT representation.
- Every vertex in the CT representation is either the root vertex or it is uniquely identified by the path to it from the root. Paths can be defined by port labels, which are invariant under pattern embeddings.
- Rooted trees are uniquely characterised by a partition of their edges into paths, which can in turn be encoded as strings. See fig. 3 for an example.

The properties of the CT representation is discussed in more details in appendix D.

[3] We can ensure that such a root always exists for example by adding dummy vertices on every edge.

Listing 1: Finding the set of canonical anchors. CANONICALANCHORS is a convenience wrapper around CONSUMEPATH, which is defined recursively. The latter returns not only the anchor list, but also the updated set of seen linear paths. `G.linear_paths(v)` is the set of all linear paths in G that go through vertex v. For traversal, a linear path `lp` is split into two paths starting at vertex v using `lp.split_at(v)`. The sequence of vertices starting from v to the end of the path is represented as a queue. The symbol ++ designates list concatenation.

```
function CanonicalAnchors(G: Graph, root: Vertex) -> List[Vertex]:
  # Initialise the variables for ConsumePath and return the anchors
  (anchors, seen_paths) = ConsumePath(G, [root], ∅):
  return anchors

function ConsumePath(
    G: Graph,
    path: Queue[Vertex],
    seen_paths: Set[LinearPath],
) → (List[Vertex], Set[LinearPath]):
  new_anchor = null
  unseen = ∅
  # Find the first vertex in the queue on an unseen linear path
  while unseen == ∅:
    if path.is_empty():
      return ([], {})
    new_anchor = path.pop()
    unseen = G.linear_paths(new_anchor) \ seen_paths

  # Add the new linear paths to the set of seen paths
  seen_paths = seen_paths ∪ unseen

  # We traverse the rest of current path as well as all the new linear paths
  paths = [path]
  for lp in unseen:
    (left_path, right_path) = lp.split_at(new_anchor)
    paths.push(left_path)
    paths.push(right_path)

  # For each path find anchors recursively and update seen paths
  anchors = [new_anchor]
  for path in paths:
    (new_anchors, new_seen_paths) = ConsumePath(G, path, seen_paths)
    anchors = anchors ++ new_anchors
    seen_paths = new_seen_paths
  return (anchors, seen_paths)
```

4.2 Pattern matching with fixed anchors

We aim to present an ℓ-independent pattern matcher: an algorithm that can identify all pattern embeddings in a subject graph (5) whose complexity is independent from the number of patterns ℓ. In this section, we restrict to pattern embeddings that map the set of canonical anchors of the pattern to a fixed subset of vertices in the subject graph, and we show that this problem can be reduced to a simple matching problem on strings.

We start by observing that such pattern embeddings with fixed anchors correspond to tree inclusions of CT representations. Let G be a port graph, let $P_1, \dots, P_\ell$ be patterns of width w and let $X \subseteq V$ be a set of w vertices in G. Choose root vertices $r_G \in X$ and r_i in patterns P_i. Write T_i for the CT representation of P_i rooted in r_i. Consider the following set $\mathscr{G}$ of subgraphs of G:[4]

$$\mathscr{G} = \{H \subseteq G \mid H \text{ is connected convex subgraph and } \textsc{CanonicalAnchors}(H, r_G) = X\}. \quad (8)$$

Proposition 6. *If $\mathscr{G} \neq \varnothing$, then there is a connected subgraph $G_{max} \subseteq G$ such that $H \subseteq G_{max}$ for all $H \in \mathscr{G}$. The split graph $split_X(G_{max})$ is a tree rooted in r_G. There is a pattern embedding $\varphi\colon P_i \to G$ mapping the canonical anchor set of P_i to X and $\varphi(r_i) = r_G$ if and only if there is an injective embedding of trees $\phi\colon T_i \to split_X(G_{max})$ with $\phi(r_i) = r_G$ that satisfies eq.* (4) *and preserves vertex labels.*

The proof gives an explicit construction for $G_{\max}$.

Proof. Let $L_1, \dots, L_w$ be the subset of linear paths in G that go through at least one vertex in X. Let $G_{\max} \subseteq G$ be the subgraph of G defined by the edges

$$E_{\max} = \bigcup_{1 \leq i \leq w} L_i, \quad (9)$$

For any subgraph $H \in \mathscr{G}$ it holds that $H \subseteq G_{\max}$ due to the linear paths of H being contained in $L_1, \dots, L_w$. Since any $H \in \mathscr{G}$ is connected, the anchors in X are connected in H and therefore also in $G_{\max}$. As a consequence, all vertices of $G_{\max}$ are connected. The port graph $\text{split}_X(G_{\max})$ must be a tree, as otherwise its canonical anchors would be a strict subset of X and by corollary 5, $width(\text{split}_X(G_{\max})) < |X|$. Hence,

$$width(G_{\max}) = width(\text{split}_X(G_{\max})) < |X| = width(H).$$

contradicting the convexity assumption of eq. (8). Assuming $\mathscr{G} \neq \varnothing$, we now prove the bidirectional implication between φ and ϕ.

$\Leftarrow$: We use vertex labels on T_i and $\text{split}_X(G_{\max})$ to mark with a unique label vertices that were split from a same vertex v in P_i, respectively $G_{\max}$ (details in appendix D). Let $\mathscr{V}_{P_i}$ and $\mathscr{V}_{G_{\max}}$ be the partition of the vertices of T_i and $\text{split}_X(G_{\max})$ into sets of split vertices with identical labels; there are bijective maps between $\mathscr{V}_{P_i}$ and the vertices in P_i as well as between $\mathscr{V}_{G_{\max}}$ and the vertices in $G_{\max}$. The tree embedding $\phi\colon T_i \to \text{split}_X(G_{\max})$ preserves vertex labels and thus maps sets in $\mathscr{V}_{P_i}$ to sets in $\mathscr{V}_{G_{\max}}$: it thus defines a map $\varphi_V : P_i \to G_{\max}$.

φ_V is injective by injectivity of ϕ and maps the root r_i to r_G by construction. The $w-1$ non-root anchor vertices of T_i are the only vertices in T_i on more than one linear path: they must be mapped to the $w-1$ vertices in $\text{split}_X(G_{\max})$ with the same properties—precisely its non-root anchor vertices. Edges are mapped bijectively by graph splitting and thus the map φ_E can be defined by using ϕ_E and must satisfy eq. (4). Since $G_{\max} \subseteq G$, we conclude that φ is a valid pattern embedding $P_i \to G$.

[4] A convex subgraph $H \subseteq G$ is one such the canonical embedding $H \to G$ is convex, as defined in assumption 3 of section 3.2.

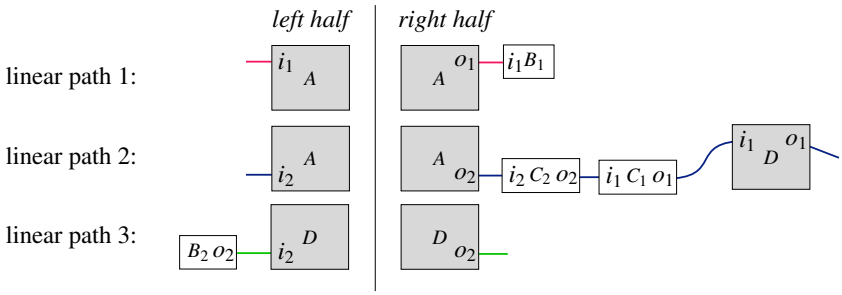


Figure 3: The split graph of fig. 2, represented by 6 strings obtained from its 3 linear paths.

$\Rightarrow$: The image of $\varphi : P_i \to G$ is a convex connected subgraph of G with canonical anchors X and root r_G, and thus is in $\mathscr{G}$. It must in particular be a subgraph of $G_{\max}$, and thus we can view φ as an embedding $\varphi : P_i \to G_{\max}$.

Note that edges are mapped bijectively between split and unsplit graphs, and thus the pattern embedding φ defines an injective map ϕ_E from edges in T_i to edges in $\text{split}_X(G_{\max})$. We construct the map $\phi : T_i \to \text{split}_X(G_{\max})$ inductively over the vertex set of T_i. We start by defining $\phi(r) := r_G$. Using eq. (4) and ϕ_E, we can then uniquely define the image of any neighbouring vertex of r in T_i. Proceeding inductively we will define ϕ on all vertices of T_i since it is connected. Because eq. (4) holds on φ, this procedure is well-defined and the resulting map ϕ will also satisfy eq. (4).

Now suppose v, v' are vertices in T_i such that $\phi(v) = \phi(v')$. By the inductive construction there are paths from the root r to v and v' respectively such that their image under ϕ_E are two paths from r_G to $\phi(v) = \phi(v')$. But $\text{split}_X(G_{\max})$ is a tree, so both paths must be equal. By bijectivity of ϕ_E, it follows $v = v'$, and thus ϕ is injective. Finally, the vertex labels are defined to be invariant under pattern embedding and thus are preserved by definition. □

Given G and a vertex set X we can thus find a maximal subgraph $G_{\max} \subseteq G$ that contains all subgraphs of G with canonical anchors X. Given $P_1, \ldots, P_\ell$, X and G, we then compute the CT representations of $P_1, \ldots, P_\ell$ and check for inclusions within $\text{split}_X(G_{\max})$. We will use an ℓ-independent tree matching algorithm for the latter task, thus solving the ℓ-independent pattern matching problem on port graphs.

String encoding of CT representations. We reduce the tree inclusion problem that results from proposition 6 to a string prefix matching problem that admits a well-known solution, discussed in proposition 14 of appendix E. The main idea is to partition the edges of the CT representation into linear paths, each of which is represented by two strings. They encode the paths from the anchor vertex on the path to either end of the linear path by expressing them as sequences of port labels. A graph of width w will have w linear paths and will be split into $2w$ strings. For the example graph of fig. 2, we obtain six split linear paths, shown in fig. 3. See appendix D for more details.

This encoding defines the $\text{AsStrings}(T, r)$ procedure: it takes as input a connected acyclic split graph T and a root r in T, and returns an encoding of T and its vertex labels as $2 \cdot \mathit{width}(T)$ strings.

Proposition 7. *Let T_1, T_2 be acyclic connected split graphs of width w and let r_1, r_2 be vertices in T_1 resp. T_2. Let $(s_1, \ldots, s_{2w}) = \text{AsStrings}(T_1, r_1)$ and $(t_1, \ldots, t_{2w}) = \text{AsStrings}(T_2, r_2)$ be the string encodings of their linear paths. Then there is an injective tree embedding $T_1 \to T_2$ that satisfies eq. (4), maps r_1 to r_2 and preserves vertex labels if and only if $s_i \subseteq t_i$ for all $1 \leq i \leq 2w$.*

The proof consists in showing that trees can be fully defined by the set of all paths from the root, which can be encoded in and recovered from the string representation. The proof is in appendix A.3.

The $\subseteq$ notations on string designates prefix inclusion. The string prefix matching problem is a simple computational task that can be generalised to to check for multiple string patterns at the same time. An overview of this problem can be found in appendix E. Putting propositions 6 and 7 together, we can thus obtain a solution for the ℓ-independent pattern matching problem for fixed anchors:

Proposition 8. *Let G be a port graph, $P_1, \dots, P_\ell$ be patterns of width w and depth at most d, and $X \subseteq V$ be a set of w vertices in G. The set of all pattern embeddings mapping the canonical anchor set of P_i to X and root r_i to r_G for $1 \leq i \leq \ell$ can be computed in time $O(w \cdot d)$ using a pre-computed prefix tree of size at most $(\ell \cdot d + 1)^w$, constructed in time complexity $O((\ell \cdot d)^w)$.* □

4.3 Enumeration of anchor sets

Assume that all patterns have at most width w and depth d. All that remains to turn proposition 8 into a complete solution for pattern matching is to enumerate all possible sets X of at most w vertices in G that are the canonical anchors of some subgraph of G of width w. The bound on the number of such sets (proposition 10) is one of the key stepping stones of this paper that makes ℓ-independent matching possible on port graphs.

Procedure. We introduce AllAnchors, a procedure similar to CanonicalAnchors of listing 1, described in listing 2 in detail. AllAnchors takes as input a port graph G, a root vertex r_G and a width $w \geq 1$, and returns all sets of w vertices that form the canonical anchors of some subgraph of G with CT representation rooted at r_G. The main difference between listings 1 and 2 is that the successive calls to ConsumePath on line 35 of listing 1 are replaced by a series of nested loops (lines 42–48 in listing 2) that exhaustively iterate over the possible outcomes for different subgraphs of G. The results of every possible combination of recursive calls are then collected into a list of anchor sets, which is returned.

Proposition 9 (Correctness of AllAnchors)**.** *Let G be a port graph and $H \subseteq G$ be a connected convex subgraph of G of width w. Let r be a vertex of H. We have* CanonicalAnchors$(H, r) \in$ AllAnchors(G, r, w).

The proof is by induction over the width w of the subgraph H and given in appendix A.4. The idea is to map every recursive call to ConsumePath in listing 1 to a call to AllConsumePath in lines 42–48 of listing 2. All recursive results are concatenated on line 47, and thus the value returned by ConsumePath will be one of the anchor sets in the list returned by AllConsumePath.

We will see that the overall runtime complexity of AllAnchors can be easily derived from a bound on the size of the returned list. For this we use the following result:

Proposition 10. *For a port graph G and vertex r_G in G, the length of the list* AllAnchors(G, r_G, w) *is in $O(c^w \cdot w^{-3/2})$, where $c = 6.75$ is a constant.*

Proof. Let C_w be an upper bound for the length of the list returned by a call to AllConsumePath for width w, and thus a bound on the length of the list returned by AllAnchors. For the base case $w = 0$, $C_0 = 1$. The returned `all_anchors` list is obtained by pushing anchor lists one by one on line 48. We can count the number of times this line is executed by multiplying the length of the lists returned by the recursive calls on lines 43–45, giving us the recursion relation

$$C_w \leq \sum_{\substack{0 \leq w_1, w_2, w_3 < w \\ w_1 + w_2 + w_3 = w-1}} C_{w_1} \cdot C_{w_2} \cdot C_{w_3}. \tag{10}$$

Listing 2: Returns all sets of w vertices that form the canonical anchors of some subgraph of G with CT representation rooted at r_G.. The code structure mirrors listing 1, with AllAnchors and AllConsumePath replacing CanonicalAnchors and ConsumePath respectively. `lp.split_at`, `G.linear_paths` and ++ are defined as in listing 1.

```
function AllAnchors(G: Graph, root: Vertex, w: Integer) -> List[List[Vertex]]:
  # Assumption: root is on a single linear path
  assert len(G.linear_paths(root)) == 1

  # Initialise the variables for AllConsumePath and return the anchor lists
  all_anchors = []
  for (anchors, seen_paths) in AllConsumePath(G, w, [root], ∅):
    all_anchors.push(anchors)
  return all_anchors

function AllConsumePath(
    G: Graph,
    w: Integer,
    path: Queue[Vertex],
    seen_paths: Set[LinearPath],
) → List[(List[Vertex], Set[LinearPath])]:
  # Base case: return one empty anchor list
  if w == 0:
    return [[]]

  new_anchor = null
  unseen = ∅
  # Find the first vertex in the queue on an unseen linear path
  while unseen == ∅:
    if path.is_empty():
      return []
    new_anchor = path.pop()
    unseen = G.linear_paths(new_anchor) \ seen_paths
  # Every vertex is on at most one unseen linear path as either
  #  - the new anchor is the root, in which case it is on at most one linear path
  #  - or it is on up to two linear paths, but one of them has already been seen.
  assert len(unseen) == 1
  new_path = unseen[0]

  # The w anchors are made of the new anchor and w-1 anchors on path1 - path3
  path1 = path
  path2, path3 = new_path.split_at(new_anchor)
  seen0 = seen_paths ∪ {new_path}
  return_list = []
  # Iterate over all ways to split w-1 anchors over the three paths
  # and solve recursively
  for 0 ≤ w1, w2, w3 < w such that w1 + w2 + w3 == w - 1:
    for (anchors1, seen1) in AllConsumePath(G, w1, path1, seen0):
      for (anchors2, seen2) in AllConsumePath(G, w2, path2, seen1):
        for (anchors3, seen3) in AllConsumePath(G, w3, path3, seen2):
          # Concatenate new anchor with anchors from all paths
          anchors = [new_anchor] ++ anchors1 ++ anchors2 ++ anchors3
          return_list.push(anchors, seen3)
  return return_list
```

Since C_w is meant to be an upper bound, we replace $\leq$ with equality in eq. (10) to obtain a recurrence relation for C_w. This recurrence relation is a generalisation of the well-known Catalan numbers [18], equivalent to counting the number of ternary trees with w internal nodes: a ternary tree with $w \geq 1$ internal nodes is made of a root along with three subtrees with w_1, w_2 and w_3 internal nodes respectively, with $w_1 + w_2 + w_3 = w - 1$. A closed form solution to this problem can be found in [1]:

$$C_w = \frac{\binom{3w}{w}}{2w+1} = \Theta\left(\frac{c^w}{w^{3/2}}\right)$$

satisfies eq. (10) with equality, where $c = {}^{27}/_{4} = 6.75$ is a constant obtained from the Stirling approximation:

$$\binom{3w}{w} = \frac{(3w)!}{(2w)!w!} = \Theta\left(\frac{1}{\sqrt{w}}\right)\left(\frac{(3w)^3}{e^3}\right)^w\left(\frac{e^2}{(2w)^2}\right)^w\left(\frac{e}{w}\right)^w = \Theta\left(\frac{({}^{27}/_{4})^w}{w^{1/2}}\right). \qquad \square$$

To obtain a runtime bound for ALLANCHORS, it is useful to identify how much of G needs to be traversed. If we suppose all patterns have at most depth d, then it immediately follows that any vertex in G that is in the image of a pattern embedding must be at most a distance d away from an anchor vertex. For this purpose, we modify the definition of `split_at` in listing 2 to only return the first d vertices of any path returned. We thus obtain the following runtime.

Corollary 11. *For patterns with at most width w and depth d, the total runtime of* ALLANCHORS *is in*

$$O\left(\frac{c^w \cdot d}{w^{1/2}}\right). \tag{11}$$

The proof is in appendix A.5. Finally, we reach our main result.

Theorem 12. *Let $P_1, \ldots, P_\ell$ be patterns with at most width w and depth d. The pre-computation runs in time and space complexity*

$$O\left((d \cdot \ell)^w \cdot \ell + \ell \cdot w^2 \cdot d\right).$$

For any subject port graph G, the pre-computed prefix tree can be used to find all pattern embeddings $P_i \to G$ in time

$$O\left(|G| \cdot \frac{c^w}{w^{1/2}} \cdot d\right) \tag{12}$$

where $c = 6.75$ is a constant.

Proof. The pre-computation consists of running the CANONICALANCHORS procedure on every pattern and then transforming them into string tuples using ASSTRINGS. ASSTRINGS is linear in pattern sizes and CANONICALANCHORS runs in $O(w^2 \cdot d)$ for each pattern (proposition 4). This is followed by the insertion of ℓ tuples of $2w$ strings of length $\Theta(d)$ into a multidimensional prefix tree. This dominates the total runtime, which can be obtained directly from proposition 14.

The complexity of pattern matching itself on the other hand is composed of two parts: the computation of all anchor set candidates, and the execution of the prefix string matcher for each of the trees resulting from these sets of fixed anchors. The complexity of the former is obtained by multiplying the result of proposition 10 with $|G|$, as ALLANCHORS must be run for every choice of root vertex r in G:

$$O(w \cdot d \cdot C_w \cdot |G|), \tag{13}$$

where C_w is the bound for the number of anchor lists returned by ALLANCHORS. For the latter we use proposition 14 and obtain the complexity $O(w \cdot d \cdot C_w)$, which is dominated by eq. (13). $\square$

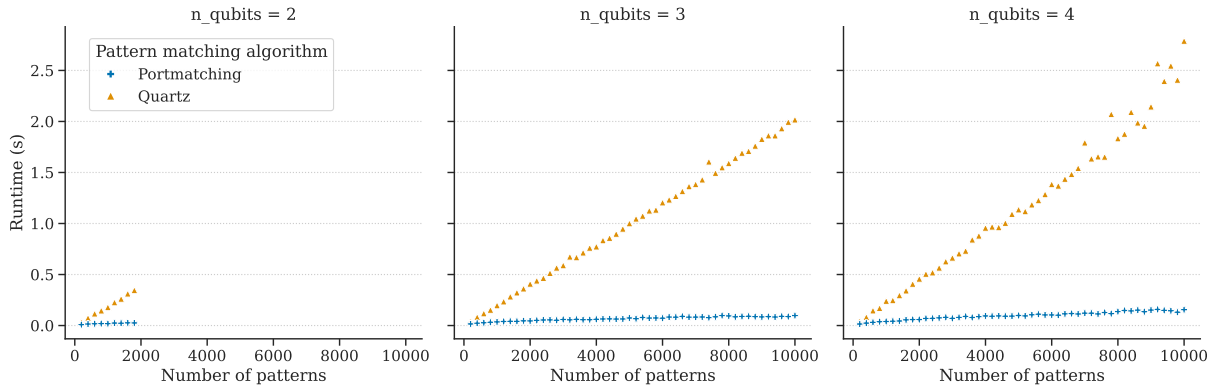


Figure 4: Runtime of pattern matching for $\ell = 0 \ldots 10^4$ patterns on 2, 3 and 4 qubit quantum circuits from the Quartz ECC dataset, for our implementation (Portmatching) and the Quartz project. All $\ell = 1954$ two qubit circuits were used, whereas for 3 and 4 qubit circuits, $\ell = 10^4$ random samples were drawn.

5 Pattern matching in practice

Theorem 12 shows that pattern independent matching can scale to large datasets of patterns but imposes some restrictions on the patterns and embeddings that can be matched. In this section we discuss these limitations and give empirical evidence that the pattern matching approach we have presented can be used on a large scale, outperforming existing solutions.

Pattern limitations. In section 3.2, we imposed conditions on the pattern embeddings in order to obtain a complexity bound for pattern independent matching. We argued how these restrictions are natural for applications in quantum computing and most of the arguments will also hold for a much broader class of computation graphs.

In future work, it would nonetheless be of theoretical interest to explore the importance of these assumptions and their impact on the complexity of the problem. As a first step towards a generalisation, our implementation and all our benchmarks in this section do not make any of these simplifying assumptions. Our results below give empirical evidence that a significant performance advantage can be obtained regardless.

Implementation. We provide an open source implementation in Rust of pattern independent matching using the results of section 4, described in more detail in appendix F. The implementation works for weighted or unweighted port graphs, and makes none of the simplifying assumptions employed in the theoretical analysis.

Benchmarks. To assess practical use, we have benchmarked our implementation against a leading C++ implementation of pattern matching for quantum circuits from the Quartz superoptimiser project [20]. Using a real world dataset of patterns obtained by the Quartz equivalence classes of circuits (ECC) generator, we measured the pattern matching runtime on a random subset of up to 10000 patterns. We considered circuits on the T, H, CX gate set with up to 6 gates and 2, 3 or 4 qubits. Thus for our patterns we have the bound $d \leq 6$ for the maximum depth and width $w = 2, 3, 4$. In all experiments the graph G subject to pattern matching was `barenco_tof_10` input, i.e. a 19 qubit circuit input with 674 gates obtained by decomposing a 10-qubit Toffoli gate using the Barenco decomposition [2]. The results are

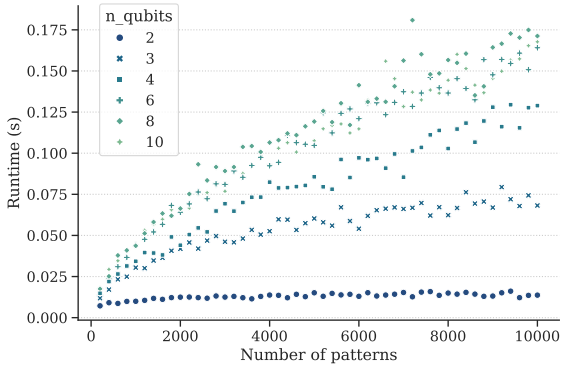


Figure 5: Runtime of our pattern matching for random quantum circuits with up to 10 qubits.

summarised in fig. 4. For $\ell = 200$ patterns, our proposed algorithm is $3\times$ faster than Quartz, scaling up to $20\times$ faster for $\ell = 10^5$.

We also provide a more detailed scaling analysis of our implementation by generating random sets of 10000 quantum circuits with 15 gates for qubit numbers between $w = 2$ and $w = 10$, using the previous gate set; the results are shown in fig. 5. From theorem 12, we expect that the pattern matching runtime is upper bounded by a ℓ-independent constant. Runtime seems indeed to saturate for $w = 2$ and $w = 3$ qubit patterns, with an observable runtime plateau at large ℓ. From the exponential c^w dependency in eq. (12), it is however to be expected that this upper bound increases rapidly for qubit counts $w \geq 4$. A runtime ceiling is not directly observable at this experiment size but the gradual decrease in the slope of the curve is consistent with the existence of the ℓ-independent upper bound predicted in theorem 12.

6 Conclusion

We have demonstrated that pattern matching on port graphs can be done in a runtime asymptotically independent of the number of patterns by pre-computing an automaton-like data structure. As a result, we obtain a provable computational advantage in the regime of numerous low-width patterns. This opens up promising avenues for graph rewriting and particularly for the optimisation of computation graphs and quantum circuits. Benchmarks further show that the approach is fast in practice. At the scale of interest (10000 pattern circuits with 3-4 qubits), the resulting implementation of pattern matching on quantum circuits is 20x times faster than that of Quartz [20], a leading quantum superoptimiser.

References

[1] Jean-Christophe Aval (2008): *Multivariate Fuss–Catalan numbers*. *Discrete Mathematics* 308(20), p. 4660–4669, doi:10.1016/j.disc.2007.08.100.

[2] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin & Harald Weinfurter (1995): *Elementary gates for quantum computation*. *Phys. Rev. A* 52, pp. 3457–3467, doi:10.1103/PhysRevA.52.3457.

[3] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski & Fabio Zanasi (2020): *String Diagram Rewrite Theory I: Rewriting with Frobenius Structure*. *Journal of the ACM (JACM)* 69, pp. 1 – 58, doi:10.1145/3502719.

[4] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski & Fabio Zanasi (2021): *String diagram rewrite theory II: Rewriting with symmetric monoidal structure*. Mathematical Structures in Computer Science 32, pp. 511 – 541, doi:10.1017/S0960129522000317.

[5] Jin Fang, Yanyan Shen, Yue Wang & Lei Chen (2020): *Optimizing DNN computation graph using graph substitutions*. In: Proceedings of the VLDB Endowment, 13, pp. 2734 – 2746, doi:10.14778/3407790.3407857.

[6] Maribel Fernández, Hélène Kirchner & Bruno Pinaud (2018): *Labelled Port Graph – A Formal Structure for Models and Computations*. Electronic Notes in Theoretical Computer Science 338, pp. 3–21, doi:10.1016/j.entcs.2018.10.002. The 12th Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2017).

[7] Charles L. Forgy (1982): *Rete: A fast algorithm for the many pattern/many object pattern match problem*. Artificial Intelligence 19(1), pp. 17–37, doi:10.1016/0004-3702(82)90020-0.

[8] Zhihao Jia, Oded Padon, James J. Thomas, Todd Warszawski, Matei A. Zaharia & Alexander Aiken (2019): *TASO: optimizing deep learning computation with automatic generation of graph substitutions*. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles, doi:10.1145/3341301.3359630.

[9] Xiaoyi Jiang & Horst Bunke (1996): *Including geometry in graph representations: A quadratic-time graph isomorphism algorithm and its applications*. In Petra Perner, Patrick Wang & Azriel Rosenfeld, editors: Advances in Structural and Syntactical Pattern Recognition, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 110–119, doi:10.1007/3-540-61577-6_12.

[10] Xiaoyi Jiang & Horst Bunke (1998): *Marked subgraph isomorphism of ordered graphs*. In Adnan Amin, Dov Dori, Pavel Pudil & Herbert Freeman, editors: Advances in Pattern Recognition, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 122–131, doi:10.1007/BFb0033230.

[11] Donald Knuth (1999): *The Art of Computer Programming: Sorting and Searching, Volume 3*. Addison-Wesley, Reading MA.

[12] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache & Oleksandr Zinenko (2021): *MLIR: Scaling Compiler Infrastructure for Domain Specific Computation*. In: 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 2–14, doi:10.1109/CGO51591.2021.9370308.

[13] Bruno T. Messmer & Horst Bunke (1999): *A decision tree approach to graph and subgraph isomorphism detection*. Pattern Recognit. 32, pp. 1979–1998.

[14] Adam Paszke, Sam Gross, Francisco Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai & S. Chintala (2019): *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. In: Neural Information Processing Systems, doi:10.5555/3454287.3455008.

[15] Xuguang Ren & Junhu Wang (2016): *Multi-Query Optimization for Subgraph Isomorphism Search*. Proc. VLDB Endow. 10(3), p. 121–132, doi:10.14778/3021924.3021929.

[16] Timos K. Sellis (1988): *Multiple-Query Optimization*. ACM Trans. Database Syst. 13(1), p. 23–52, doi:10.1145/42201.42203. Available at `https://doi.org/10.1145/42201.42203`.

[17] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington & Ross Duncan (2020): *tket: a retargetable compiler for NISQ devices*. Quantum Science and Technology 6(1), p. 014003, doi:10.1088/2058-9565/ab8e92.

[18] Richard P. Stanley (2015): *Catalan Numbers*. Cambridge University Press, doi:10.1017/CBO9781139871495.

[19] Amanda Xu, Abtin Molavi, Lauren Pick, Swamit Tannu & Aws Albarghouthi (2023): *Synthesizing Quantum-Circuit Optimizers*. Proc. ACM Program. Lang. 7(PLDI), doi:10.1145/3591254.

[20] Mingkuan Xu, Zikun Li, Oded Padon, S. Lin, J. Pointing, A. Hirth, H. Ma, J. Palsberg, A. Aiken, U.A. Acar & Z. Jia (2022): *Quartz: Superoptimization of Quantum Circuits*. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2022, Association for Computing Machinery, New York, NY, USA, p. 625–640, doi:10.1145/3519939.3523433.

A Proofs

A.1 Proof of proposition 2

Proposition 2. *Let G be a port graph with n_{odd} vertices of odd degree and n_ω open ports. Then the graph width of G is at most $\lfloor (n_{odd} + n_\omega)/2 \rfloor$.*

Proof. For any acyclic linear path $P \subseteq E^*$ in G consider its two endpoints v_1 and v_2, i.e. the two vertices in G that are only incident to one edge in P (linear paths are never empty). Let p_1 and p_2 be the ports where the first and last edges are attached to v_1 and v_2 respectively. Let $p_i' \in ports(v_i)$ such that $p_i' \sim p_i$ for $i = 1,2$. By eq. (3), either $\lambda(v_i, p_i') \in P$ or $\lambda(v_i, p_i') = \omega$. By injectivity of $\lambda(v_i, \cdot)$, the first case implies $p_i = p_i'$ as we would otherwise have two edges $\lambda(v_i, p_i') \neq \lambda(v_i, p_i)$ in P incident to v_i. We thus conclude that either $\lambda(v_i, p_i') = \omega$ or p_i is in a singleton equivalence class of $\sim$ in $ports(v_i)$.

There are n_ω pairs $(v, p) \in V(G) \times \mathscr{P}$ such that $\lambda(v_i, p_i') = \omega$ and n_{odd} pairs $(v, p) \in V(G) \times \mathscr{P}$ such that the equivalence class of p is a singleton set in $ports(v)$. We conclude that there can be at most $n_{\text{odd}} + n_\omega$ end ports of linear paths in G. As every linear path has two end ports and every end port must be distinct, the result follows. □

A.2 Proof of proposition 4

Proposition 4. *For a connected port graph G of width w and depth d and for a vertex r of G, listing 1 returns an anchor set of w vertices; we call this the set of canonical anchors. Its runtime is $O(w^2 \cdot d)$.*

Proof. Termination: we count the number of times CONSUMEPATH is called in one execution of CANONICALANCHORS. The call on line 3 happens exactly once, so we can ignore it. On the other hand, the `path` argument to CONSUMEPATH will always be distinct between any two calls on line 33: it is either a path on a previously unseen linear path, or it is a strict subset of the `path` argument passed to the current call. As there are w linear path with at most d vertices, there is a finite number of calls to CONSUMEPATH. The while loop on lines 14–18 pops an element from the `path` queue at each iteration, so can only be executed a finite number of times. Thus we can conclude that the CONSUMEPATH procedure always terminates.

Correctness: CANONICALANCHORS returns w vertices: in every call to CONSUMEPATH, the only non-recursive insertion to the list of anchors is the initialisation of the `anchors` list on line 31. This insertion happens if and only if `unseen` is non-empty (line 14). Using the assumption that every vertex is on at most 2 linear paths, we can furthermore restrict ourselves to $|\texttt{unseen}| = 1$. Thus the size of `seen_paths` increases by one at every recursion (line 21). The size of `seen_paths` is bounded by w and thus w anchors are added to the `anchors` list over the execution of CANONICALANCHORS.

Let X be the vertices returned by CANONICALANCHORS as a set. It remains to be shown that the X-split graph of G is connected and acyclic. A cycle C in $\text{split}_X(G)$ must have edges on at least 2 distinct linear paths by the second assumption of section 3.2. Say there are $k > 2$ distinct linear paths on the cycle. Every anchor vertex on the cycle can be on either 1 or 2 linear paths (we assumed in section 3.2 that no vertex is on more than two paths). There must be at least k anchor vertices on C whose two adjacent edges that are also in C are on two different linear paths—one for every "switch" of linear path on C. However, by line 14, for every anchor there is at least one unseen linear path, so for k anchors there must be at least $k+1$ linear paths in C, which is impossible.

For connectedness, observe that for every vertex v in the split graph there is a path from the root r to v. In G, such a path is obtained by following the graph traversal implicit in the calls to CONSUMEPATH:

let $\tilde{v}$ be the vertex in G that when split generates v. Every vertex in G appears in the `path` argument to CONSUMEPATH at least once. There is thus an anchor $a \in X$ with a path along a linear path from a to $\tilde{v}$. Applying this argument recursively, there is a sequence of anchors $r = a_1, a_2, \ldots, a_k = \tilde{v}$ corresponding to successive calls to CONSUMEPATH such that for all $1 \leq i < k$ there is a linear path between a_i and a_{i+1}.

We show that the path from r to $\tilde{v}$ through $a_1, \ldots, a_k$ is mapped onto a path in the split graph. In other words, we need to show that the edges along the path are rewired in such a way that adjacent edges along the path are mapped to edges adjacent to the same split vertex. We partition the path into sections from a_i to a_{i+1} for $i = 1, \ldots, k-1$ and consider each subpath separately. Let $e_1, \ldots, e_m$ be the edges of the subpath from a_i to a_{i+1}. The first edge e_1 on this subpath is always in the split graph as $a_i \in X$ and thus is not split. Every other edge, on the other hand, is on the same linear path as e_1. Thus for $1 \leq j < m$, if e_j ends in port p and the next edge e_{j+1} starts in port p', then $p \sim p'$. Thus both edges are mapped to the same split vertex, concluding that the path from r to v is also a path in the split graph.

Complexity: Note that a recursive call to CONSUMEPATH (line 33) occurs if and only if the current call is adding a new element to the list of anchor vertices (line 31). Since we have previously established that the number of anchor vertices returned by CANONICALANCHORS is w, it follows that there are at most w recursive calls to CONSUMEPATH. Therefore, to prove the runtime complexity $O(w^2 \cdot d)$ of CANONICALANCHORS, it remains to show that the execution of the body of CONSUMEPATH—excluding line 33—runs in $O(w \cdot d)$. This is straightforward to check for all lines but 18 and 26. Line 26 is executed at most w times on a single call to CONSUMEPATH, and since `lp.split_at(v)` simply needs to traverse a linear path of at most length d, the required runtime complexity holds. On the other hand, line 18 is executed at most d times on a single call to CONSUMEPATH—since that is what it takes for line 17 to pop all elements from the path.

Assuming the list of linear paths was computed in advance (in time $O(w \cdot d)$, before the first call to CONSUMEPATH in line 3) and each linear path is given a unique index $0 \ldots w-1$, we can store the `seen_paths` set and the set of linear paths for each vertex as ordered lists of linear path indices. The corresponding set `G.linear_paths(v)` can be stored as an attribute of `v` and be retrieved in constant time (assuming for instance that vertices are indexed from 0 to $|G|-1$). Other than that, line 18 is a set operation that can be realised in a single $O(w)$ pass over the ordered lists `unseen_paths` and `G.linear_paths(v)` since both have at most w elements. □

A.3 Proof of proposition 7

Proposition 7. *Let T_1, T_2 be acyclic connected split graphs of width w and let r_1, r_2 be vertices in T_1 resp. T_2. Let $(s_1, \ldots, s_{2w}) = \text{ASSTRINGS}(T_1, r_1)$ and $(t_1, \ldots, t_{2w}) = \text{ASSTRINGS}(T_2, r_2)$ be the string encodings of their linear paths. Then there is an injective tree embedding $T_1 \to T_2$ that satisfies eq.* (4)*, maps r_1 to r_2 and preserves vertex labels if and only if $s_i \subseteq t_i$ for all $1 \leq i \leq 2w$.*

Proof. The $\Rightarrow$ direction is straightforward. By assumption, the root r_1 in T_1 is mapped to the root r_2 in T_2. Non-root anchors on the other hand are precisely the vertices on more than one path in T_1 and T_2. If $\varphi : T_1 \to T_2$ is an injection of trees of the same width, then all non-root anchors of T_1 must be mapped to the non-root anchors of T_2. Every linear path in T_1 includes at least one anchor vertex. This must be mapped by φ to a path in T_2, through the corresponding anchor vertex in T_2. As φ preserves the port labels (eq. (4)), the image path in T_2 must be a subpath of a linear path of T_2. As the linear paths are split and ordered starting from anchor vertices, the string encoding of every split linear path in T_1 will be a prefix of the string encodings of split linear paths in T_2.

$\Leftarrow$: it suffices to show that every path from root to a vertex in T_1 is also a path from root to a vertex in T_2 and that the vertex labels coincides. A path P from root r_1 in T_1 can be partitioned into a sequence of paths $P = P_1 \cdots P_k$, which all start at anchors and are subpaths of linear paths of T_1. These subpaths corresponds to a sequence of prefixes of $s_{\alpha_1}, s_{\alpha_k}$ in the string encoding, which are also prefixes of $t_{\alpha_1}, \ldots, t_{\alpha_k}$. Since the vertex labels are stored in the tuple string encoding, we know that the end vertex of the path $P_1 \cdots P_i$ coincides with the anchor of $t_{\alpha_{i+1}}$ in T_2. Applying this argument recursively on the chain of linear subpaths, we conclude that $P = P_1 \cdots P_k$ is also a path in T_2. Finally, the vertex labels must coincide on the shared domain of definition, as the string encoding coincide. Equation (4) can be shown to be satisfied using a similar argument to the one presented in the proof of proposition 6. □

A.4 Proof of proposition 9

Proposition 9 (Correctness of ALLANCHORS)**.** *Let G be a port graph and $H \subseteq G$ be a connected convex subgraph of G of width w. Let r be a vertex of H. We have* CANONICALANCHORS$(H, r) \in$ ALLANCHORS(G, r, w).

Proof. Let $H \subseteq G$ be a connected subgraph of G of width w. We prove inductively over w that

$$\text{CONSUMEPATH}(H, \texttt{path}, \texttt{seen\_paths}) \in \text{ALLCONSUMEPATH}(G, w, \texttt{path}, \texttt{seen\_paths}) \tag{14}$$

for all arguments `path` and `seen_paths`. The statement in the proposition follows from this claim directly.

For the base case $w = 1$, CONSUMEPATH will return `[new_anchor]`, where `new_anchor` is obtained from lines 16–20 of listing 1: there is only one linear path and thus for every recursive call to CONSUMEPATH, `unseen` will be empty, until `path` has been exhausted and the empty list is returned. The definition of `new_anchor` coincides with the one obtained from lines 20–28 of listing 2. The only values of `w1`, `w2` and `w3` that satisfy the loop condition on line 42 of listing 2 for $w = 1$ are $\texttt{w1} = \texttt{w2} = \texttt{w3} = 0$. Using the base condition on lines 18–20 of listing 2, we conclude that ALLCONSUMEPATH$(G, 1, \texttt{path}, \texttt{seen_paths})$ returns `[[new_anchor]]`, satisfying eq. (14).

We now prove the claim for $w > 1$ by induction. Using our simplifying assumptions, we obtain the assertion on line 32 of listing 2, as documented. For listing 1, this assumption simplifies the loop on lines 34–37 to at most three calls to CONSUMEPATH with arguments (H, P_{curr}, S_{curr}), (H, P_ℓ, S_ℓ) and (H, P_r, S_r) respectively, where

- P_{curr} is the value of the `path` variable after line 20,
- P_ℓ and P_r refer to the two halves of the new linear path, as computed and stored in the variables `left_path` and `right_path` on line 28, and
- S_{curr}, S_ℓ and S_r are the values of the `seen_paths` variable after the successive updates on line 23 and two iterations of line 37.

Consider a call to CONSUMEPATH (listing 1) with arguments $\texttt{G} = H$ and some variables `path` and `seen_paths`. Let w_{curr}, w_ℓ and w_r be the length of the values returned by the three recursive calls to CONSUMEPATH of line 35. As every anchor vertex reduces the number of unseen linear paths by exactly one (using the simplifying assumptions), it must hold that $w_{curr} + w_\ell + w_r + 1 = w$. Thus for a call to ALLCONSUMEPATH (listing 2) with arguments $\texttt{G} = G$, $\texttt{w} = w$ and the same values for `path` and `seen_paths`, there is an iteration of the `for` loop on line 42 of listing 2 such that $\texttt{w1} = w_{curr}, \texttt{w2} = w_\ell$ and $\texttt{w3} = w_r$. The definition of `seen0` on line 38 of listing 2 coincides with the update to `seen_paths` on line 23 of listing 1; it follows that on line 43 of listing 2 the recursive call ALLCONSUMEPATH$(G, w_{curr}, P_{curr}, S_{curr})$ is

executed. From the induction hypothesis we obtain that there is an iteration of the `for` loop on line 43 of listing 2 in which `anchors1` and `seen1` coincide with the `new_anchors` and `new_seen_paths` variables of the first iteration of the `for` loop on line 34 of listing 1. In particular the value of `seen1` is equal S_ℓ.

Repeating the argument, we obtain that there are iterations of the `for` loops on lines 44 and 45 of listing 2 that correspond to the second and third calls to CONSUMEPATH on line 35 of listing 1. Finally, the concatenation of anchor lists on line 47 of listing 2 is equivalent to the repeated concatenations on line 36 of listing 1 and so we conclude that eq. (14) holds for w. □

A.5 Proof of corollary 11

Corollary 11. *For patterns with at most width w and depth d, the total runtime of* ALLANCHORS *is in*

$$O\left(\frac{c^w \cdot d}{w^{1/2}}\right). \tag{11}$$

Proof. We restrict `split_at` on line 37 to only return the first d vertices on the linear path in each direction: vertices more than distance d away from the anchor cannot be part of a pattern of depth d.

We use the bound on the length of the list returned by calls to ALLCONSUMEPATH of proposition 10 to bound the runtime. We can ignore the non-constant runtime of the concatenation of the outputs of recursive calls on line 47, as the total size of the outputs is asymptotically at worst of the same complexity as the runtime of the recursive calls themselves. Excluding the recursive calls, the only remaining lines of ALLCONSUMEPATH that are not executed in constant time are the `while` loop on lines 24–28 and the `split_at` call on line 37.

Consider the recursion tree of ALLCONSUMEPATH, i.e. the tree in which the nodes are the recursive calls to ALLCONSUMEPATH and the children are the executions spawned by the nested `for` loops on line 42–48. This tree has at most

$$C_w = \Theta\left(\frac{c^w}{w^{3/2}}\right)$$

leaves. A path from the root to a leaf corresponds to a stack of recursive calls to ALLCONSUMEPATH. Along this recursion path, the `seen_paths` set is always strictly growing (line 38) and the vertices popped from the `path` queue on line 27 are all distinct. `split_at` is called once for each of the w linear path that are added to `seen_paths`. For each linear path two paths of length at most d are traversed and returned. Thus the total runtime of `split_at` along a path from root to leaf in the recursion tree is in $O(w \cdot d)$. Similarly, the number of executions of the lines 25–28 is bound by the number of elements that were added to a `path` queue, as for every iteration an element is popped off the queue on line 27. This is equal to the number of elements returned by `split_at`, resulting in the same complexity. We can thus bound the overall complexity of executing the entire recursion tree by $O(C_w \cdot w \cdot d) = O(\frac{c^w \cdot d}{w^{1/2}})$. □

B Lower bound on the number of patterns

Proposition 13. *Let $N_{w,d}$ be the number of port graphs of width w, depth d and maximum degree $\Delta \geq 4$. We can lower bound*

$$N_{w,d} > \left(\frac{w}{2e}\right)^{\Theta(wd)},$$

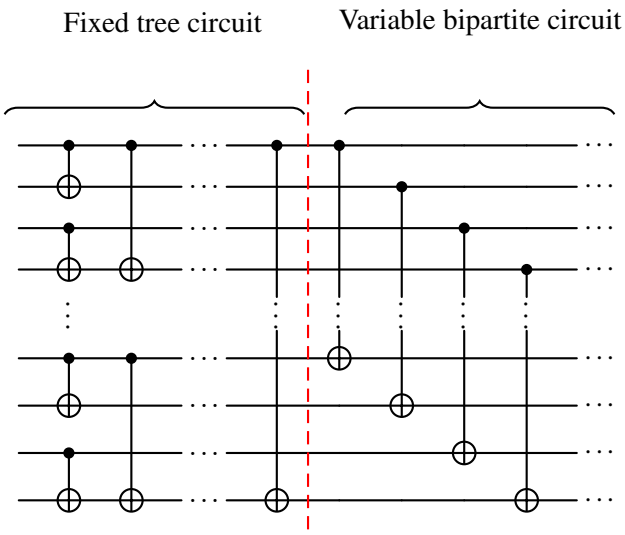
assuming $w \leq o(2^d)$.

In the regime of interest, w is small, so the assumption $w \leq o(2^d)$ is not a restriction. In the main text we use the bound $|P| \leq w \cdot d$ to avoid introducing the circuit depth. The bound stated in eq. (2) is thus slightly looser.

Proof. Let $w, d > 0$ and $\Delta \geq 4$ be integers. We wish to lower bound the number of port graphs of depth d, width w and maximum degree Δ. It is sufficient to consider a restricted subset of such port graphs, whose size can be easily lower bounded. We will count a subset of CX quantum circuits, i.e. circuits with only *CX* gates, a two-qubit non-symmetric gate. Because we are using a single gate type, this is equivalent to counting a subset of port graphs with vertices of degree 4. Assume w.l.o.g that w is a power of two. We consider CX circuits constructed from two circuits with w qubits composed in sequence:

- **Fixed tree circuit**: A $\log_2(w)$-depth circuit that connects qubits pairwise in such a way that the resulting port graph is connected. We fix such a tree-like circuit and use the same circuit for all CX circuits. We can use this common structure to fix an ordering of the w qubits, that refer to as qubits $1, \ldots, w$.
- **Bipartite circuit**: A CX circuit of depth $D = d - \log_2(w)$ with exactly $w/2 \cdot (d - \log_2(w))$ CX gates, each gate acting on a qubit $1 \leq q_1 \leq w/2$ and a qubit $w/2 < q_2 \leq w$.

The following circuit illustrates the construction:



All that remains is to count the number of such bipartite circuits. Every slice of depth 1 must have $w/2$ CX gates acting on distinct qubits. Every qubit 1 to $w/2$ must interact with one of the qubits $w/2+1$ to w, so there are $(w/2)!$ such depth 1 slices. Repeating this depth 1 construction D times and using Sterling's approximation, we obtain a lower bound for the number of port graphs of depth d, width w and maximum degree at least 4:

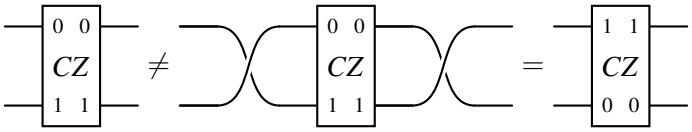
$$\left(\left(\frac{w}{2}\right)!\right)^D > \sqrt{w\pi}\left(\frac{w}{2e}\right)^{wD/2} = \left(\frac{w}{2e}\right)^{\Theta(w\cdot d)}$$

where we used $w = o(2^d)$ to obtain $\Theta(D) = \Theta(d)$ in the last step. □

C Quantum circuits as port graphs

A relevant consideration when viewing quantum circuits as port graphs is the question of equality on circuits. We consider two circuits to be equal if they are equal as port graphs. This sense of equality is more general than equality of ordered lists of gates, another common internal representation of quantum computations, but does not account for commuting gates or gate *symmetries*. An example of a symmetric gate type is the CZ gate, a gate type of arity $n = 2$ that is symmetric in its arguments

that is to say, exchanging the order of the inputs and outputs does not change the computation. Viewed as port graphs, however, the left and right hand side are distinct circuits



In the case that such symmetries need to be taken into account for pattern matching, there are two simple solutions. For rewriting purposes, one may choose to add a single rewrite rule to express the symmetry explicitly, stating that the symmetric gate can be rewritten to itself with the edge order reversed. This will recover the full expressivity of the rewrite rule set, at the expense of additional rewrite rule applications.

Alternatively, all instances of a pattern that are equivalent up to gate symmetries can be enumerated and added as separate patterns to the matcher. This approach is particularly appealing as the runtime of the pattern matcher will remain unchanged, despite the increase in the number of patterns (exponential in the number of symmetric gates). The trade-off is increased pre-compilation time and pattern matcher size.

D Properties of the Canonical Tree representation

We provide here the exact derivations of the properties of the CT representation that we rely on, namely an injective map from the port graph representation to CTs, invariance of the CT representation under pattern embeddings and the string encoding of CT trees.

Equivalence of the CT representation. A connected port graph G is fully defined by the set of edges, given as a set of pairs in $V \times \mathscr{P}$. Given the CT representation of G with vertices $\tilde{V}$, alongside a map $merge : \tilde{V} \to V$ that maps the vertices of the CT representation to the vertices of G, it is immediate that G can be recovered by mapping every $(v, p) \in \tilde{V} \times \mathscr{P}$ to $(merge(v), p) \in V \times \mathscr{P}$.

Up to isomorphism in the co-domain V, we can store *merge* by storing the partition of $\tilde{V}$ into sets with the same image. We introduce for this a map $\tilde{V} \to \tilde{V}$ that maps every vertex to a canonical representative of the partition—for instance the vertex closest to the root in CT. This map can be stored as vertex labels of CT, which we can refer to as the labelled CT representation for distinction. However in the main text, it is always the labelled representation that is meant when CT representations are discussed.

We thus have a bijective map between the labelled CT representation of G and the port graph G. Furthermore, this map preserves the linear paths, i.e. it maps one to one the linear paths of the labelled CT representation to the linear paths of G. For all purposes, we can thus treat the labelled CT representation as an equivalent representation of G.

Invariance under pattern embedding. Unlike graphs, rooted trees can be defined in a way that is invariant under bijective relabelling of the vertices by using the invariant port labels. Every tree vertex is either the root vertex or it is uniquely identified by the path to it from the root. Since paths can be defined in terms of port labels, paths are invariant under pattern embeddings of the underlying graph.

For trees T and T', let S and S' be the sets of their respective vertices expressed as sequences of port labels. We thus define tree inclusion and equality only up to vertex relabelling: $T \subseteq T'$ if and only if $S \subseteq S'$, and $T = T'$ if and only if $S = S'$. On labelled trees, we also require inclusion (resp. equality) of the vertex label maps, including in particular the *merge* map of labelled CT representations. As a result, subtree relations in labelled CT representations correspond to subgraphs of the original graph, in effect reducing the pattern matching problem on port graphs to a problem of tree inclusion on CT representations. This statement is formalised in proposition 6.

String encoding of CT representations. In order for our string encoding of CT representations to map tree inclusion to string prefixes, we recall that the anchor set in eq. (8) is fixed: a subtree of T with the same anchor set can only be obtained by shortening the linear paths at their ends— the resulting subpath will always contain the anchor vertex. Given a linear path L of T, we thus split L at the anchor on L and obtain two paths L_1, L_2 starting from the anchor to the ends of L. For any subtree $T' \subseteq T$, the linear path L' that is a subpath of L will split into L'_1, L'_2, prefixes of L_1 and L_2 respectively.

With an appropriate string representation of CT vertices and their labels, this will encode all linear paths. In the same way that the *merge* map of the labelled CT representation is used to restore the original graph from the split CT vertices, we use it to recover the anchor vertices from the split linear paths. Finally, to order the linear paths in the string tuple, we use for instance the order of their anchors induced by port ordering.

E Prefix Trees

Our main result is achieved by reducing a tree inclusion problem to the following problem.

String prefix matching. Consider the following computational problem over strings. Let Σ be a finite alphabet and consider $\mathscr{W} = (\Sigma^*)^w$ the set of w-tuples of strings over Σ. For a string tuple $(s_1, \ldots, s_w) \in \mathscr{W}$ and a set of string tuples $\mathscr{D} \subseteq \mathscr{W}$, the w-dimensional string prefix matching consists in finding the set

$$\{(p_1, \ldots, p_w) \in \mathscr{D} \mid \text{for all } 1 \leq i \leq w : p_i \text{ is a prefix of } s_i\}.$$

This string problem can be solved using a w-dimensional prefix tree. We give a short introduction to prefix trees for the string case but refer to standard literature for more details [11].

One-dimensional prefix tree. Let $P_1, \ldots, P_\ell \in \mathscr{A}^*$ be strings on some alphabet $\mathscr{A}$. Given an input string $s \in \mathscr{A}^*$, we wish to find the set of patterns $\{P_{1 \leq i \leq \ell} | P_i \subseteq s\}$, i.e. P_i is a prefix of s.

The prefix tree of $P_1, \ldots, P_\ell$ is a tree with a tree node for each prefix of a pattern. The children of an internal node are the strings that extend the prefix by one character. The root of the tree is the empty string. Each tree node also stores a list of matching patterns, with each pattern stored in the unique corresponding node. Every prefix tree has an empty string node, which is the root of the tree. For every inserted pattern of length at most L nodes are inserted, one for every non-empty prefix of the pattern. Thus a one-dimensional prefix tree has at most $\ell \cdot L + 1$ nodes and can be constructed in time $O(\ell \cdot L)$.

Given an input $s \in \mathscr{A}^*$, we can find the set of matching patterns by traversing the prefix tree of $P_1, \ldots, P_\ell$ starting from the root. We report the list of matching patterns at the current node and move to

the child node that is still a prefix of s, if it exists. This procedure continues until no more such child exists. In total the traversal takes time $O(|s|)$, as every character of s is visited at most once.

Note that in theory the number of reported pattern matches can dominate the runtime of the algorithm. We can avoid this by returning the list of matches as an iterator, stored as a list of pointers to the tree nodes matching lists.

Multi-dimensional prefix tree. A w-dimensional prefix tree for $w > 1$ is defined recursively as a one-dimensional prefix tree that at each node stores a $w-1$-dimensional prefix tree. Given an input w-tuple $(s_1, \ldots, s_w) \in (\mathscr{A}^*)^w$, the traversal of the w-dimensional prefix tree is done by traversing the one-dimensional prefix tree on the input s_1 until no child is a prefix of the input, and then recursively traversing the $w-1$-dimensional prefix tree on $(s_2, \ldots, s_w)$. Similarly to the one-dimensional case, the list of matching patterns is stored at prefix tree nodes and reported during traversal. The traversal thus takes time $O(|s_1| + \cdots + |s_w|)$, as every character of s is visited at most once.

For ℓ tuples of size w of words of maximum length L, we can bound the number of nodes of the w-dimensional prefix tree by $1 + (\ell \cdot L)^w$. The runtime and space complexity of the construction of the w-dimensional prefix tree is thus in $O((\ell \cdot L)^w)$, summarised in the result:

Proposition 14. *Let $\mathscr{D} \subseteq \mathscr{W}$ be a set of string tuples and L the maximum length of a string in a tuple of $\mathscr{D}$. There is a prefix tree with at most $(\ell \cdot L)^w + 1$ nodes that encodes $\mathscr{D}$ that can be used to solve the w-dimensional string prefix matching problem in time $O(|s_1| + \cdots + |s_w|)$.*

F Open source implementation

The code is available at `https://github.com/lmondada/portmatching/`. All benchmarking can be reproduced using the tooling and instructions at `https://github.com/lmondada/portmatching-benchmarking`.

We represent all the pattern matching logic within a generalised finite state automaton, composed of states and transitions. This formalism is used to traverse the graph input and express both the prefix tree of the string prefix matching problem and the (implicit) recursion tree of listing 2 in section 4.3. We sketch here the automaton definition. Further implementation details can be obtained from the `portmatching` project directly.

In the pre-computation step, the automaton is constructed based on the set of patterns to be matched. It is then saved to the disk; a run of the automaton on an input graph G is the solution the pattern independent matching problem for the input G. To run the automaton, we keep track of the set of current states, initialised to a singleton root state and updated following allowed transitions from one of the current states. Which transitions are allowed is computed using predicates on the input graph stored at the transitions. This is repeated until no further allowed transitions exist from a current state.

At any one state of the automaton, zero, one or several transitions may be allowed depending on the input graph. As the automaton is run for a given input graph G, we keep track of the vertices that have been matched by the automaton so far with an injective map between a set of unique symbols and the vertices of G. Vertices in this map are the known vertices of G. There are three main types of transitions:

- A **constraint** transition asserts that a property of the known vertices holds. This can be checking for a vertex or edge label, or checking that an edge between two known vertices and ports exists.
- A **new vertex** transition asserts that there is an edge between a known vertex v at a port p and a new vertex at a port p'. The new vertex must not be any of the known vertices. When the transition is followed, a new symbol is introduced and the vertex is added to the symbol vertex map.

- A **set anchor** transition is an ε-transition, i.e. a non-deterministic transition that is always allowed. Semantically, it designates a known vertex as an anchor.

By requiring that all constraint transitions from a given state assert mutually exclusive predicates (such as edges starting from a given vertex and port, or the vertex label of a given vertex), we can ensure that constraint transitions are always deterministic. New vertex transitions are also deterministic in finite depth patterns [5], so that in the regime explored in this paper, the only source of non-determinism is the choice of anchors. Intuitively, this corresponds to the facts that the prefix tree traversal of section 4 is deterministic while the anchors enumeration of listing 2 returns a multitude of options to be explored exhaustively.

To obtain a set of matching patterns from a run of the automaton, we store pattern matches as lists at the automaton states. When a state is added to the set of current states, its list of matches are added to the output. To build the automaton, we consider one pattern at a time, convert it into a chain of transitions of the above types that is then added to the state transition graph. At the target state of the last transition, we then add the pattern ID to the list of matched patterns.

[5] In cyclic and non-convex cases, it can happen that a vertex is both a known vertex of a large pattern and a new vertex within a smaller subpattern.