Quantum Information and Computation, Vol. 24, No. 5&6 (2024) 0361–0410

QUANTUM ALGORITHMS FOR COMMUNITY DETECTION AND THEIR EMPIRICAL RUN-TIMES

CHRIS CADE[1a] MARTEN FOLKERTSMA[2] IDO NIESEN[1] JORDI WEGGEMANS[2]

[1] *QuSoft & University of Amsterdam (UvA), Amsterdam, the Netherlands*
[2] *QuSoft & CWI, Amsterdam, the Netherlands*

Received Augest 4, 2023
Revised March 18, 2024

We apply recent work [9] on empirical estimates of quantum speedups to the practical task of community detection in complex networks. We design several quantum variants of a popular classical algorithm – the *Louvain algorithm* for community detection – and first study their complexities in the usual way, before analysing their complexities empirically across a variety of artificial and real inputs. We find that this analysis yields insights not available to us via the asymptotic analysis, further emphasising the utility in such an empirical approach. In particular, we observe that a complicated quantum algorithm with a large asymptotic speedup might not be the fastest algorithm in practice, and that a simple quantum algorithm with a modest speedup might in fact be the one that performs best. Moreover, we repeatedly find that overheads such as those arising from the need to amplify the success probabilities of quantum sub-routines such as Grover search can nullify any speedup that might have been suggested by a theoretical worst- or expected-case analysis.

Keywords: Quantum heuristic algorithm, Community detection, Use case query estimation, Quantum benchmarking, Discrete optimization, Quantum graph algorithms, Grover search

1 Introduction

Estimating the impact that a quantum computer could have for a given computational problem requires an honest assessment of the potential improvement in speed[b] that a quantum algorithm might achieve over the state-of-the-art classical one. This is a tricky task made harder by the current level of maturity of quantum hardware: quantum algorithms can only be implemented for very small problem instances, and even then the output is so marred by noise that it is often difficult to assess even the correctness of the computation. As such, one often resorts to theoretical analyses of quantum algorithms and proves, rigorously, that they are likely to achieve a speedup over an equivalent classical algorithm, assuming perhaps that overheads such as those from error correction are suitably modest. Such analyses usually yield upper bounds on the worst-case run-times of the quantum algorithms – very often for artificially constructed, 'difficult' problem instances – and a speedup is concluded whenever

[a] Present address: Fermioniq, Amsterdam, the Netherlands
[b] or, indeed, accuracy.

these bounds scale better than those obtained via an analysis of the best-known classical algorithm.

Whilst this approach offers valuable insight, it does not tell the entire story. For instance, it can be the case that a provable run-time is not available, or that it is but ends up being uninformative, as is commonly the case for heuristic algorithms that have large worst-case run-times, but often perform well on instance of practical interest. Moreover, for a particular computational problem it might be possible to design several variants of a quantum algorithm, all with run-times that scale differently in different situations, and it can be difficult to decide which one will be fastest in practice. Often the performance of such algorithms will depend on the particular inputs on which they are run, something that can be somewhat difficult to account for in a mathematical study. Moreover, it can be the case that a quantum version of a classical algorithm only speeds-up part of the algorithm, and how much of an overall speedup can be obtained is input dependent, and not clear from the asymptotic behaviour.

As such, it is likely that it will be necessary to study the performance of quantum algorithms from a more empirical standpoint in order to assess their usefulness in the time before large, fault tolerant quantum computers become widespread. In [9], we suggested a general framework for doing so that was based on the combination of carefully derived upper bounds on the complexities of quantum sub-routines with classical simulation of the entire algorithm, in a way that allowed for the estimation of the quantum run-time. We gave evidence for the utility of this approach by studying the potential quantum speed-ups that might arise from quantum versions of classical heuristic algorithms for solving MAXSAT, an optimisation problem that generalises the SAT problem.

In this paper, we apply the methodology developed in [9] to a problem of more practical interest. We consider quantum speedups of a popular heuristic algorithm that goes by the name of the *Louvain algorithm*[c] which forms one of the main tools for tackling a problem ubiquitous in the study of complex networks: that of *community detection*. Together with its descendants, the Louvain algorithm has successfully been used to study large sparse networks with millions of vertices [6, 11, 26, 30]. Taking this as a use-case, we demonstrate further the usefulness of such a 'semi-empirical' approach in both estimating the potential for quantum speedups, as well as in the *design* of quantum algorithms for a particular problem. In particular, we show how a numerical study can unveil significant performance differences between various quantum algorithms for the same problem, that were not obvious a priori from an asymptotic analysis alone.

Community detection One of the main topics in the study of complex networks is whether the nodes in the network form densely connected clusters, called communities. Uncovering the community structure of a network allows for a better understanding of the network as a whole. The task of partitioning the network into communities is known as community detection. Community detection plays an important role in a variety of topics, such as (but by no means limited to) social networks [25], recommendation systems [1], E-commerce [29], scientometrics [19], biological systems and healthcare [31], and economics [15].

[c]The algorithm takes the name of the city in which it was developed. The original paper describing the method has been cited over 15,000 times, and the algorithm itself can be found in all popular graph/network analysis software packages.

Community detection falls under the broader category of graph partitioning: formally, community detection is the task of partitioning the vertex set of a graph by maximizing a particular function that expresses the quality of the partition. The most commonly used metric is the *modularity*, which assigns a value between -1 and 1 to each partition of the vertex set of the graph [14] by assigning an effective weight[d] to each edge and then summing the effective weights of all edges connecting vertices that are in the same community. The number of communities is not fixed ahead of time, and hence finding how many partitions are needed to maximise the modularity is included as a part of the problem.

Complexity-wise, the problem of finding the partition that exactly maximises the modularity is an NP-hard problem [8], and therefore it is common (indeed, necessary) to resort to heuristic methods for community detection. Commonly used heuristics are those based on hierarchical agglomeration [10] extremal optimisation [12], simulated annealing [27, 16], spectral algorithms [24], and the Louvain method [6]. The Louvain algorithm has been found to be one of the fastest and best performing algorithms in various comparative analyses [20, 32], and it is this algorithm that we choose as the basis for our quantum algorithms for community detection.

1.1 Summary of results

We design several quantum algorithms for community detection on graphs by building quantum versions of the Louvain algorithm, and analyse their complexities in the usual way, finding, as is not uncommon for heuristic quantum algorithms, a per-step speedup. We then apply the bounds and methodology from [9] to estimate the complexities of the algorithms on practical inputs, and investigate whether the speedups promised by the asymptotic analyses manifest in practice.

Our results are summarised in Table 1, in which we compare the expected complexities of the classical Louvain algorithm, as well as three quantum variants of it: '**QLouvain**' (a direct speedup of the classical algorithm), '**SimpleQLouvain**' (a simplification of the preceding algorithm), and '**EdgeQLouvain**' (a quantum version of a significant simplification of the Louvain algorithm). We also consider 'sparse graph' versions of the algorithms (indicated by a '**SG**' suffix)), whose asymptotic run-times are worse than their original versions, but whose run-times in practice are likely to be faster when the input graph is sparse. The two right-most columns provide information about the *empirically observed* speedups obtained by the algorithms when they were simulated, in the sense discussed above.

One source of overhead that we find contributed significantly to the overall complexities were logarithmic overheads due to success probability amplification of subroutines – for the algorithms to work correctly, we often require that *all* calls to quantum subroutines succeed with high probability, which often yields quite large overheads in practice. Because of this, we found that in general the more complicated quantum algorithms offered less of a speedup (or none at all) in practice, despite indicating a generic square-root speedup per step over the original classical algorithm. These observations suggest that 'greedily' favouring a larger asymptotic speedup might actually lead to *slower* run-times in practice, and that a more nuanced analysis is required if we are to maximise quantum speedups in practice.

[d] The actual edge weight minus the expected weight in the so-called *configuration model*.

	Query complexity per step k	Absolute speed-up observed?	Empirically observed range of polynomial speed-ups
Louvain	$\mathcal{O}\left(\delta_{\max}\tau_k\right)$	–	–
QLouvain	$\tilde{\mathcal{O}}(\sqrt{\delta_{\max}\tau_k})$	No	0.85 - 0.99
QLouvainSG	$\tilde{\mathcal{O}}(\delta_{\max}\sqrt{\tau_k})$	No	0.70 - 0.86
SimpleQLouvain	$\tilde{\mathcal{O}}\left(\sqrt{\frac{\delta_{\max}}{f_k}}\right)$	No	1.04 - 1.25
SimpleQLouvainSG	$\tilde{\mathcal{O}}(\frac{\delta_{\max}}{\sqrt{f_k}})$	No	1.13 - 1.55
VTAA QLouvain	$\tilde{\mathcal{O}}(t^q_{\text{avg}}\sqrt{\tau_k})$	–	–
EdgeQLouvain	$\tilde{\mathcal{O}}\left(\frac{1}{\sqrt{h_k}}\right)$	Yes	1.18 - 1.49

Table 1. Overview of the main results obtained by applying our techniques to the proposed quantum versions of the Louvain algorithm. The second column shows upper bounds on the expected number of queries when performing a single step. The third column indicates whether we observed an absolute query count speed-up by the quantum algorithm over Louvain on our artificially generated networks up to size $n = 10^5$, and the fourth column shows the estimated range of polynomial speed-ups based on the same data. Here, $\delta_{\max}$ is the maximum number of communities adjacent to any single vertex, f_k is the fraction of vertices in the graph that are 'good' during step k, τ_k is the number of vertices inspected by the classical algorithm during step k, h_k is the fraction of *edges* and $\tilde{h}_k$ the fraction of node-neighbouring community pairs that yield good moves for vertices during step k. t^q_{avg} is defined in Section 3.2.4. For a definition of all these terms, we defer to Section 3 of the paper.

For us, these findings underscore the need to investigate and consider the *actual* quantum speedup that might be achieved in practice on realistic data sets, rather than concluding that a speedup will be obtained from an asymptotic analysis alone. As we show in this paper, such an approach can also be quite useful for comparing different quantum versions of the same algorithm, something that could facilitate the future design of quantum algorithms for practical tasks.

1.2 Methodology

In [9], we introduced the necessary tools and methodology for obtaining accurate numerical estimates of the complexities of quantum algorithms with a reasonably generic form that is common to many quantum speedups of classical heuristic algorithms. In particular, we considered algorithms with the form shown in Algorithm 1. Our approach was to run the

Algorithm 1 Generic quantum algorithm structure

1: **Input** X, **Memory** M
2: **for** $k = 1, \ldots, T$ **do**
3: Do some classical processing on X and M, resulting in some list L_k containing t_k marked items.
4: Perform either one or more (perhaps nested) Grover searches with an unknown number of marked items on L_k, or run quantum maximum-finding on the list L_k, to obtain some item x_k.
5: Do some more classical processing given x_k, update M.
6: **end for**

algorithm classically, by replacing Step 4 (the call to a quantum sub-routine) with a classical

procedure that gives the same output behaviour whilst collecting the information required to estimate what the quantum run-time complexity would have been if it had been used. The quantum complexity estimates themselves were obtained via tight bounds, including all constants, of two important sub-routines: Grover search with an unknown number of marked items, and quantum maximum finding. In cases where the information required to calculate the quantum complexities could not be computed exactly (e.g. because the input sizes were too large to make a such a computation infeasible), we gave methods for estimating them whilst retaining guarantees on the complexities produced.

In all cases, 'complexity' refers to a particular choice of measure, which for us was (and will be) the number of times a particular function is called by the classical or quantum algorithm. Of course, this does not represent the true run-time, and in particular does not include overheads such as those from quantum error correction. Nevertheless, this choice of complexity enables a clean comparison between classical and quantum algorithms, as well as between different quantum algorithms for the same task. It might be that a quantum speedup suggested by our (empirical) analysis does not manifest in practice due to such overheads, but that is not our main focus – our goal is to study whether a quantum speedup could manifest *at all*, even assuming zero overhead from the likes of error correction or noise. If the algorithms fall short at this level of analysis, then a quantum speedup can already be ruled out without taking the time and effort to compile the quantum algorithm for a particular piece of hardware. In addition, this complexity measure is independent of the details of quantum hardware, which is likely to change over the coming years.

Organization

In Section 2 we introduce the practical task of community detection, describe the popular classical Louvain algorithm for it (Section 2.2), and analyse its asymptotic complexity (Section 2.3). In Section 3 we proceed to construct several quantum variants of the Louvain algorithm, with the aim of comparing the classical and quantum performances empirically. In Section 4, we give tight bounds on the complexities of our main quantum sub-routines (Section 4.1), and describe our approach to simulating these quantum algorithms (Section 4.2). Finally, in Section 5 we analyse their complexities numerically, comparing their performances to the original classical algorithm, and amongst each other.

Appendix 1 discusses the number of vertices moved by the Louvain algorithm, both from a theoretical and an empirical perspective; Appendix B provides some numerical results comparing the performance of the original Louvain algorithm with our implementation thereof that contains and additional data structure; Appendix C gives details of a slightly more efficient quantum algorithm for community detection based on the technique of variable time amplitude amplification; and finally Appendix D describes the algorithm we use to generate FCS-type random graphs.

2 Community detection

In this section, we formally introduce the problem of community detection in graphs and describe the Louvain algorithm. We begin by introducing some notation, and then proceed to define the *modularity* function, which serves as a measure of quality for community assignments, before describing the Louvain algorithm itself.

Notation In this manuscript, $G = (V, E)$ is a graph with vertex set V and edge set E. We write $n = |V|$ for the number of vertices and denote the $n \times n$ (weighted) adjacency matrix of the graph by A, which we assume to be symmetric, $A = A^T$, real-valued with non-negative entries, and without self-loops: $A_{vv} = 0$ for all $v \in V$. We write d_u for the degree of a vertex $u \in V$ and $s_u = \sum_v A_{uv}$ for the *strength* of vertex u, defined as sum of the weights of all edges incident to u. We denote the neighborhood of u by $N_u := \{v \in V : A_{uv} > 0\}$. Furthermore, write $d_{\max} = \max_{u \in V} d_u$ for the maximum degree, and let $W = \frac{1}{2}\sum_{uv} A_{uv}$ be the sum of all weights. Finally, for any positive integer k, we write $[k]$ for the set $\{1, \ldots, k\}$.

Access to the input graph We assume we have *adjacency list access* to the graph G. That is, for each $u \in V$, we have access to the list of neighbors of u through the function $\mathfrak{n}_u : [d_u] \to V$. Specifically, given $j \in [d_u]$, we can query the j-th neighbor $\mathfrak{n}_u(j) \in N_u \subseteq V$ of u, as well as the weight $A_{u\mathfrak{n}_u(j)}$ on the edge connecting u and $\mathfrak{n}_u(j)$. We assume that we know the degrees of each vertex ahead of time, or otherwise that we compute them during the pre-processing step (see below).

Finally, we assume that the vertices have some arbitrary but fixed ordering, and that the adjacency lists are sorted according to this ordering, so that, given any vertex $u \in V$ and a neighbor $v \in N_u$, we can in $\mathcal{O}(\log d_u)$ time find the index $j \in [d_u]$ such that $\mathfrak{n}_u(j) = v$ using binary search. If the adjacency lists are not sorted when they are given to use, then instead we can sort them all in time $\tilde{O}(nd_{\max})$ before continuing.

2.1 Modularity

Formally, a community partitioning of V is given by a label function $\ell : V \to [n]$ that assigns to every vertex $v \in V$ a label $\ell(v) \in [n]$. All vertices with the same label are said to be in the same community, and we denote the community of a given vertex $v \in V$ by $C_{\ell(v)} \subset V$. Likewise, for any label $\alpha \in [n]$, $C_\alpha := \ell^{-1}(\alpha)$ denotes the set of all vertices contained in the community labelled α. For clarity we will use Roman characters (e.g. u, v) to refer to vertices, and Greek letters (e.g. α, β) for community labels.

Given a community assignment ℓ, the *modularity* is defined as

$$Q := \frac{1}{2W} \sum_{u,v \in V} \left(A_{uv} - \frac{s_u s_v}{2W} \right) \delta^\ell(u, v) = \frac{1}{2} \sum_{u,v \in V} Q_{uv} \delta^\ell(u, v), \tag{1}$$

where

$$\delta^\ell(u, v) = \begin{cases} 1 & \text{if} \quad \ell(u) = \ell(v) \\ 0 & \text{otherwise} \end{cases}$$

and we write

$$Q_{uv} := \frac{1}{W} \left(A_{uv} - \frac{s_u s_v}{2W} \right).$$

Note that, like A, Q is also symmetric: $Q_{uv} = Q_{vu}$. $Q = Q(\ell)$ in Eq. (1) should be thought of as a function of ℓ; however, we will suppress the ℓ dependence of Q unless it is ambiguous as to which ℓ we are referring.

For our purposes it will be more convenient to express Q as

$$Q = \sum_{u<v} Q_{uv} \delta^\ell(u, v) + \frac{1}{2} \sum_{u \in V} Q_{uu} \tag{2}$$

where the second term is a constant independent of the label function ℓ. Since our objective is to find an ℓ that maximizes Q, we can safely ignore the second (constant) term in Eq. (2).

For a vertex $u \in V$, we call a community C_α a *neighboring community* of u if $C_\alpha \cap N_u \neq \emptyset$. The Louvain algorithm only moves vertices to neighboring communities. For a vertex u we write

$$\zeta_u := \{\alpha \in [n] : C_\alpha \cap N_u \neq \emptyset\}$$

for the set of labels of communities that neighbour u, and

$$\delta_u = |\zeta_u|$$

for the number of neighboring communities of u. In addition,let

$$S_u^\alpha := \sum_{v \in C_\alpha} A_{uv}, \quad \text{and} \quad \Sigma_\alpha := \sum_{v \in C_\alpha} s_v,$$

i.e. S_u^α is the sum of all weights on edges from vertex u to vertices in community C_α, and Σ_α is the sum of all weights on edges incident to vertices contained in community C_α. (Note that in the expression for S_u^α we can actually restrict the sum over all C_α to $N_u \cap C_\alpha$, since $A_{uv} = 0$ for all $v \in V$ not neighboring u.)

The Louvain algorithm attempts to move vertices from one community to the next in a greedy way by only making moves that strictly increase the modularity. Suppose that a vertex u currently in community $C_{\ell(u)}$ is moved to neighboring community C_α. Then the change in the modularity Δ_u^α resulting from this move is given by

$$\begin{aligned}
\Delta_u^\alpha &= \sum_{w \in C_\alpha} Q_{uw} - \sum_{w \in C_{\ell(u)} \setminus \{u\}} Q_{uw} \\
&= \frac{1}{W} \sum_{w \in C_\alpha} A_{uw} - \frac{s_u}{2W^2} \sum_{w \in C_\alpha} s_w - \frac{1}{W} \sum_{w \in C_{\ell(u)}} A_{uw} + \frac{s_u}{2W^2} \sum_{w \in C_{\ell(u)} \setminus \{u\}} s_w \\
&= \frac{S_u^\alpha - S_u^{\ell(u)}}{W} - \frac{s_u \left(\Sigma_\alpha - \Sigma_{\ell(u)} + s_u\right)}{2W^2},
\end{aligned} \tag{3}$$

where we have used that $A_{uu} = 0$. Finally, for a fixed vertex u, we define $\bar{\Delta}_u := \max_{\alpha \in \zeta_\alpha} \Delta_u^\alpha$. Note that both $\Delta_u^\alpha = \Delta_u^\alpha(\ell)$ and $\bar{\Delta}_u = \bar{\Delta}_u(\ell)$ depend on ℓ, but we will again suppress the ℓ-dependence unless it is relevant for the statement in question.

2.2 The Louvain algorithm

The Louvain algorithm alternates between two *phases*. The first phase consists of a number of greedy moves that attempt to increase modularity. When there are no more moves left to make, the second phase contracts communities into single vertices, and then the whole process repeats itself at this new coarse-grained level. The two phases repeat until, at some point, no new moves exist directly at the start of a phase 1.

Because we will introduce several (quantum) versions of the Louvain algorithm, we will refer to the Louvain algorithm from [6] as the *original Louvain* algorithm (discussed below), or OL for short. For a precise description of the algorithm, see Algorithm 2.

Algorithm 2 The Louvain algorithm

function LOUVAIN(Graph G, Community set $\mathcal{C}$)
  $\mathcal{C} \leftarrow$ SINGLEPARTITION(G) ▷ assign each node its own community
  done $\leftarrow$ False
  while not done **do**
    $\mathcal{C}' \leftarrow$ MOVENODES(G, $\mathcal{C}$) ▷ get new community assignment
    done $\leftarrow \|\mathcal{C}\| = \|V\|$ ▷ end when every community consists of one node
    if not done **then**
      G $\leftarrow$ AGGREGATEGRAPH(G, $\mathcal{C}$)
      $\mathcal{C} \leftarrow$ SINGLEPARTITION(G)
    end if
  end while
  return $\mathcal{C}$
end function

function MOVENODES(Graph G, Community set $\mathcal{C}$)
  done $\leftarrow$ False
  while not done **do**
    done $\leftarrow$ True
    for all $u \in V$ **do**
      $\bar{\Delta}_u \leftarrow \max_{v \in N_u} \Delta_{uv}$ ▷ calculate maximum increase of modularity
      if $\bar{\Delta}_u > 0$ **then**
        $\bar{v} \leftarrow \arg\max_{v \in N_u} \Delta_{uv}$ ▷ get corresponding community
        $\ell(u) \leftarrow \ell(\bar{v})$ ▷ reassign u to community of $\bar{v}$
        done $\leftarrow$ False ▷ terminate when there is no modularity increase
      end if
    end for
  end while
end function

function AGGREGATEGRAPH(Graph G, Community set $\mathcal{C}$)
  $V' \leftarrow \{C_a \| C_a \neq \emptyset\}$ ▷ create new vertex for every nonempty set
  $A' \leftarrow \{A'_{ab} | a, b \in V', A'_{ab} = \sum_{u \in C_a, v \in C_b} A_{uv}\}$
  $E' \leftarrow \{(a, b) | a, b \in V', A'_{ab} > 0\}$
  ▷ create edges with weight equal to the sum of all weights between vertices in each community
  return GRAPH(V', E', A')
end function

function SINGLEPARTITION(Graph G)
  return $\{\{v\} | v \in V\}$
end function

Initialization

Initially every vertex is assigned to its own community $\ell(u) = u$. Before beginning, for every $u \in V$, loop over all neighbors $j \in [d_u]$ in order to compute the vertex strengths s_u as well as each $\Sigma_{\ell(u)} = s_u$, and also the total edge weight sum $W = \frac{1}{2} \sum_{uv} A_{uv}$. If not already sorted, we also sort all adjacency lists during initialization.

First phase

During the first phase, the algorithm places all vertices in a randomly ordered list. This list is traversed sequentially and, for each vertex encountered, we compute $\bar{\Delta}_u = \max_{\alpha \in \zeta_u} \Delta_u^\alpha$. If $\bar{\Delta}_u > 0$, u is moved to the community that realises $\arg\max_{\alpha \in \zeta_a} \Delta_u^\alpha$. After completing a pass through the list, it is reshuffled and the process is repeated. This phase ends when there are no vertices that can be moved to increase the modularity any further, i.e. when $\bar{\Delta}_u \leq 0$ for all $u \in V$.

In order to compute $\bar{\Delta}_u$ for a given a $u \in V$, the algorithm can first construct the list of neighboring community labels ζ_u as well as a list $L_u = \{(\alpha, S_u^\alpha) : \alpha \in \zeta_u\}$ of neighboring communities and corresponding sums of edge weights from u to those communities. Now a single loop over L_u is sufficient to compute Δ_u^α for every $(\alpha, S_u^\alpha) \in L_u$ and output $\bar{\Delta}_u$ and $\bar{\alpha} = \arg\max_{\alpha \in \zeta_u} \Delta_u^\alpha$. If $\bar{\Delta}_u > 0$, then u is moved from its original community to the new community $C_{\bar{\alpha}}$.

As vertices move from one community to the next, the algorithm maintains a list of the sums $\{\Sigma_\alpha : \alpha \in [n]\}$. In particular, after moving vertex u from its original community C_β to its new community $C_{\bar{\alpha}}$, we subtract s_u from Σ_β and add it to $\Sigma_{\bar{\alpha}}$ to ensure that the quantities $\{\Sigma_\alpha : \alpha \in [n]\}$ are kept up to date. The algorithm then also updates the label function ℓ.

Second phase

After the first phase has finished and no vertex move can further increase the modularity, a new coarse-grained graph $G' = (V', E')$ with weighted adjacency matrix A' is constructed. This new coarse-grained graph has as its vertex set the set of (non-empty) communities constructed in the first phase: $V' = \{C_\alpha : \alpha \in [n], C_\alpha \neq \emptyset\}$. An edge is present in G' between two vertices $C_\alpha, C_\beta \in V'$ if there is an edge between any two vertices in corresponding communities in G, i.e. $E' = \{(C_\alpha, C_\beta) : \exists u \in C_\alpha, v \in C_\beta \text{ such that } (u, v) \in E\}$, and its weight is the sum of all the edge weights of edges between C_α and C_β in G, i.e. $A'_{\alpha\beta} = \sum_{u \in C_\alpha, v \in C_\beta} A_{uv}$. Constructing V', E' and A' can be done in a single loop over all edges of G.

2.3 Complexity of the Louvain algorithm

As discussed, the goal of this paper is to compare the performance of the Louvain algorithm to its quantum counterparts using the empirical method outlined in the introduction, and in doing so to tackle some of the obstacles and considerations that one might face in taking such an approach. As we discussed there, this means that we must choose a measure of complexity for our algorithms to use to make our (empirical) comparisons. Concretely, we choose to count the number of calls to the function that computes the change in modularity resulting from moving a vertex from one community to another. In this section we first precisely define our complexity measure, and then analyse the complexity of the classical Louvain algorithm.

2.3.1 Complexity measure

We consider how many calls are made to the function that computes the change in modularity resulting from a particular vertex move. I.e. for a particular vertex u and community α that

it might move to, we count calls to (an oracle that computes) the function[e]

$$g_\Delta(s_u, \Sigma_\alpha, \Sigma_{\ell(u)}, S_u^\alpha, S_u^{\ell(u)}) = \frac{S_u^\alpha - S_u^{\ell(u)}}{W} - \frac{s_u\left(\Sigma_\alpha - \Sigma_{\ell(u)} - s_u\right)}{2W^2}\,. \tag{4}$$

Note that this can be seen counting the number calls to (the gradient of) the modularity Q that we are attempting to maximize, which is often the natural measure of complexity for optimization algorithms.

Counting the number of function calls does not capture every part of the algorithms' complexities. Recall that the classical Louvain algorithm consist of several phases, in which the initialization phase does not require any function calls and the second (coarse-graining) phase similarly does not, even though both require a single loop over all m edges. By taking the number of function calls as a means of comparing the quantum and classical algorithms, we are inherently not taking into account the initialization and second phases. However, in practice the first phase takes up the vast majority of the computation time, and moreover the initialization and second phases are identical for both the classical and quantum algorithms, and hence ignoring them in our comparisons is sensible. Another aspect that is not measured by the number of function calls is the time it takes to compute the list L_u for each vertex u considered. In Section 2.3.2 below, we argue that the classical algorithm can be improved upon slightly by keeping all of these lists in memory, and only updating those that change after moving a vertex. The time it takes to update this list is also not captured in the comparison between the quantum and classical algorithms, however, as with the initialization and second phases, these updates involve the same operations for both the quantum and classical algorithms.[f]

Instead of using the number of function calls to the modularity function, a common complexity measure for graph algorithms is the number of queries to the graph. However, the initialization and second phases of the algorithm already require us to query all edges of the input graph. Beyond this no further queries need to be made, since we can simply query all edges once and then store the results in memory, and thus all of our algorithms have 'query complexity' $|E|$. Hence, to meaningfully compare the algorithms we consider in this work empirically, we will simply count the number of calls to the function g_Δ.

In the sections that follow we will also consider the *time/gate complexity* of the algorithms, in terms of how many additional elementary operations are required besides the calls to g_Δ. This will be useful in order to compare the worst-case asymptotic run-times of the classical and quantum algorithms, but for the purposes of numerical comparison it is much cleaner to focus only on calls to g_Δ, since the precise number (i.e. including constants) of elementary gates required for various operations quickly becomes architecture-dependent.

[e] When we perform our numerical study in Section 4, we will count calls to the function g_Δ, and each oracle call will correspond to $c_q = 2$ function calls.

[f] We would have liked to include the time it takes to update the lists also in our analysis. However, working out the precise complexity of doing so is impossible without assuming a particular quantum hardware architecture. In particular, it necessitates the introduction of several new architecture-dependent variables that encompass how read and write times compare between classical and quantum memories, and how these times compare to the cost of computing the function g_Δ. Including these quantities as tune-able parameters in the analysis would have made things less clear. By choosing to count only the number of function calls as a means of comparison, we have chosen not to focus on the part of the algorithm that involves updating data structures.

2.3.2 Complexity of Louvain

For every vertex u visited during the steps of the first phase, there are δ_u calls to g_Δ (one for each community α adjacent to u) required to compute $\bar{\Delta}_u$ and $\bar{\alpha}$, as well as $O(d_u)$ other operations needed to construct the list L_u. The total complexity then depends on how many vertices are visited in the entire first phase of the algorithm. If we suppose the algorithm makes T moves in total, and that on the kth move it must inspect t_k vertices before finding one that it can move, then the total number of function calls (queries) required by the algorithm will be

$$\sum_{k\in[T]} O\left(\delta_{\max} t_k\right) ,$$

and the number of other operations

$$\sum_{k\in[T]} O\left((\delta_{\max} + d_{\max}) t_k\right) .$$

Since the algorithm is heuristic, it is difficult to accurately bound the total number of moves T. In Appendix A.1 we show that, in general, T can be upper bounded by a polynomial in n, and hence the Louvain algorithm is always a polynomial-time algorithm. In practice, however, T often scales as $O(n \log n)$ [20] – as confirmed also by our numerical data presented in Appendix A.2.

The run-time (but not the number of function calls) of the original classical algorithm can be improved slightly at the expense of a constant overhead in space complexity, by making use of an additional data structure. Recall that the classical algorithm computes, for each vertex u that it visits, a list $L_u = \{(\alpha, S_u^\alpha) : \alpha \in \zeta_u\}$, and then uses these values (plus the Σ_α's and s_u's also stored in memory) as input to the function g_Δ. This list takes $\tilde{O}(d_u)$ time to construct (since we must loop over all neighbours of u to compute the appropriate sums), leading to $\tilde{O}((\delta_{\max} + d_{\max})t_k)$ time required for the kth step. We can improve this complexity if we store the information contained in L_u for every u separately, and update it as appropriate.

Data structure In particular, for each $u \in V$ we introduce a 'community adjacency list' $\eta_u : [\delta_u] \to [n] \times \mathbb{R}$, which given an index j, returns the label α of the jth neighbouring community to u, as well as the sum S_u^α. As shorthand we will often write $\alpha = \eta(j)$, even though $\eta(j)$ actually returns the tuple (α, S_u^α). We will keep the list η_u sorted by community label (according to some arbitrary but fixed ordering), allowing lookup of S_u^α using label α in $O(\log \delta_u)$ time. Finally, we will reserve a special place in this list to store the quantity $S_u^{\ell(u)}$, and assume that we can access this directly. We will refer to the sets $\{s_u : u \in V\}$, $\{\Sigma_\alpha : \alpha \in [n]\}$ and $\{\eta_u : u \in V\}$ collectively as the *data structure*.

The data structure therefore allows us to obtain the inputs to g_Δ all in constant time. Now we concern ourselves with the time required to update it. Suppose that we move vertex u from community α to community β. In terms of the $\Sigma_\cdot$ values, it is clear that only Σ_α and Σ_β change. These are easily updated by subtracting s_u from the former and adding it to the latter, which requires $O(\log n)$ time. The only community adjacency lists η_v that will change will be for vertices v that are neighbours of u: since each entry in any η_v stores only sums of weights of edges incident to neighbouring communities of v, any sum that doesn't include

an edge to u will remain unchanged. Within each η_v (for $v \in N_u$), the only sums that will change will be the ones corresponding to the communities that have changed: namely, S_v^α and S_v^β. The list η_v is sorted by community label, and so we can identify the indices i and j corresponding to communities α and β in $O(\log d_{\max})$ time each using binary search. Then we update the tuple $\eta_v(i) = (\alpha, S_v^\alpha)$ by subtracting A_{uv} from S_v^α, and we update the tuple $\eta_v(j) = (\beta, S_v^\beta)$ by adding A_{uv} to S_v^β, where each operation will take time $O(\log d_{\max})$. If we find that the new value of S_v^α is equal to zero, we remove that tuple from the list, and if the tuple (β, S_v^β) does not already exist, then we create it and insert it into the list at its sorted position. Note that we will only add a new tuple if the list is not already of length d_v, and hence the length of the list remains less than or equal to d_v.

For each neighbour v of u, these updates therefore take time $O(\log d_{\max})$. Since we do this for every neighbour of u, the total time for all updates is $O(d_u \log_{\max}) \leq O(d_{\max} \log d_{\max})$.

Complexity with the data structure By using the data structure described above, we can eliminate the need to construct the list L_u for each vertex u, at the cost of having to update the data structure after every move. In this case, the number of function calls remains the same, but the classical algorithm now takes time

$$\sum_{k=1}^{T} \tilde{O}\left(\delta_{\max} t_k + d_{\max}\right) .$$

We verify numerically in Appendix B that the addition of the data structure does indeed improve the run-time of the algorithm.

3 Quantum algorithms for community detection

In this section we present a number of quantum variants of the original Louvain algorithm for community detection. Our reason for introducing several quantum algorithms is to later study, in Section 4, how much of the promised asymptotic (per-step) speedup actually manifests in practice for different variants of the algorithm, and to demonstrate how an empirical comparison between algorithms can reveal significant differences in their run-times that aren't made clear by an asymptotic analysis alone. However, in this section we will only concern ourselves with the usual kind of asymptotic analysis of algorithm complexity.

We begin by introducing a quantum algorithm that mimics the classical algorithm exactly (i.e. by searching for the first good vertex from a randomly ordered list of vertices), and which makes use of a nested Grover search. We then construct a much-simplified variant that forgoes the ordered list and directly applies a nested Grover search to the entire set of vertices. Both of these algorithms also make use of quantum maximum finding to obtain the *best* move available to a particular good vertex. In the end these algorithms are somewhat sub-optimal: the nested Grover searches are performed over sets of varying sizes, but the outer Grover search complexity is limited by the size of the largest set, something that does not happen in the classical case. To overcome this drawback, in Section 3.2.4 we introduce a slightly more sophisticated quantum algorithm that makes use of the technique of *variable time amplitude amplification*, which allows the subroutine called by a Grover search to have different stopping times. However, since we do not numerically study this algorithm, we defer its details to Appendix C. Later in the section we consider dropping the nested Grover search

format all together in favour of an asymptotically sub-optimal, but likely practically more efficient, implementation that instead makes use of a classical subroutine, and which might be much more efficient on sparse input graphs. Finally, we present a much-simplified quantum algorithm that performs a single Grover search over the space of *edges* of the graph, in search of one that suggests a good move.

3.1 Quantum preliminaries

We will find the following quantum subroutines useful. Later, in Section 4, we will consider explicit implementations of them as given in [9], and take into account their full run-times (i.e. including all constants). For **QSearch** (Lemma 1 below), this will in particular mean including an extra argument (N_{samples}) to the algorithm that determines how many classical samples are drawn before Grover search is used, but which does not affect the asymptotic runtime.

Lemma 1 (Grover's search with an unknown number of marked items [7]) *Let L be a list of items, and t the (unknown) number of 'marked items'. Let $\mathcal{O}_g |x_i\rangle |0\rangle = |x_i\rangle |g(x_i)\rangle$ be an oracle that provides access to the Boolean function $g : [|L|] \to \{0,1\}$ that labels the items in the list. Then there exists a quantum algorithm* ***QSearch****(L, ϵ) that finds and returns an index i such that $g(x_i) = 1$ with probability at least $1-\epsilon$ if one exists and requires an expected number $O(\sqrt{N/t}\log(1/\epsilon))$ queries to $\mathcal{O}_g$ and $O(\sqrt{N/t}\log(N/\epsilon))$ other elementary operations. If no such x_i exists, the algorithm confirms this and to do so requires $O(\sqrt{N}\log(1/\epsilon))$ queries to $\mathcal{O}_g$ and $O(\sqrt{N}\log(N/\epsilon))$ other elementary operations.*

Lemma 2 (Exact Grover search [18]) *Let L be a list of items, and $t > 0$ the* known *number of 'marked items'. Let $\mathcal{O}_g |x_i\rangle |0\rangle = |x_i\rangle |g(x_i)\rangle$ be an oracle that provides access to the Boolean function $g : [|L|] \to \{0,1\}$ that labels the items in the list. Then there exists a quantum algorithm* ***ExactQSearch****(L, t) that finds and returns an index i such that $g(x_i) = 1$ with* certainty*. To do so, the algorithm makes $O(\sqrt{N/t})$ queries to $\mathcal{O}_g$ and $O(\sqrt{N}\log(N))$ other elementary operations.*

Lemma 3 (Quantum maximum-finding [13]) *Let L be a list of items of length $|L|$, with each item in the list taking a value in the interval $[a, b]$, to which we have coherent access in the form of a unitary that acts on basis states as*

$$\mathcal{O}_L |x\rangle |0\rangle = |x\rangle |L[x]\rangle .$$

Then there exists a quantum algorithm ***QMax****(L, ϵ) that will return $\arg\max_x L[x]$ with probability at least $1-\epsilon$ using at most $O(\sqrt{|L|}\log(1/\epsilon)$ queries to $\mathcal{O}_f$ (i.e. to the list L) and $O(\sqrt{|L|}\log|L|\log(1/\epsilon))$ elementary operations.*

We will also make use of the variable time amplitude amplification (VTAA) algorithm of Ambainis [3]. The statement of this result is somewhat more involved, and so we will defer to Appendix C for a more formal description of VTAA and its run time in the context of our particular application of it, and discuss the technique informally here.

Consider a quantum algorithm $\mathcal{A}$ which may stop at one of several times $t_1, \ldots, t_m$. To indicate the outcome, $\mathcal{A}$ has an extra register O with 3 possible values 0, 1, and 2: 0 indicates that the computation has stopped but did not reach the desired outcome; 1 indicates that the computation has stopped and the desired outcome was reached; 2 indicates that the

computation has not stopped yet. The idea behind VTAA is to run multiple branches of computation in superposition, and to amplify those branches that have either stopped and reached the desired outcome (1) (e.g. found a marked item), or are still running (2).

Let p_i be the probability of the algorithm stopping at time t_i (with either the outcome 0 or outcome 1). The average stopping time of $\mathcal{A}$ (the l_2 average) is

$$T_{\text{avg}} := \sqrt{\sum_i p_i t_i^2}. \tag{5}$$

Let $T_{\max} = t_m$ be the maximum possible running time of $\mathcal{A}$,

$$\alpha_{\text{good}} |1\rangle_O |\psi_{\text{good}}\rangle + \alpha_{\text{bad}} |0\rangle_O |\psi_{\text{bad}}\rangle$$

be the final state of the algorithm once all branches have stopped, and $p_{\text{succ}} = |\alpha_{\text{good}}|^2$ be the probability of obtaining the state $|\psi_{\text{good}}\rangle$ using algorithm $\mathcal{A}$. Then Ambainis [3] shows the following.

Lemma 4 (Variable time amplitude amplification [3]) *There exists a quantum algorithm $\mathcal{A}'$ invoking $\mathcal{A}$ several times, for total time*

$$\tilde{O}\left(T_{\max} \log(T_{\max}) + \frac{T_{avg}}{\sqrt{p_{succ}}} \log^{1.5} T_{\max}\right)$$

that produces a state $\alpha |1\rangle |\psi_{good}\rangle + \beta |0\rangle |\psi'\rangle$ such that $|\alpha|^2 > 1/2$. By repeating $\mathcal{A}'$ $O(\log \frac{1}{\epsilon})$ times, we can obtain $|\psi_{good}\rangle$ with probability at least $1 - \epsilon$.

This is in contrast to the usual amplitude amplification routine, which would take time $O(T_{\max}/\sqrt{p_{\text{succ}}})$, and hence we see a speedup whenever T_{avg} is substantially smaller than $T_{\max}$. However, the algorithm $\mathcal{A}$ must satisfy a number of constraints (in particular, it cannot be adaptive), and so VTAA is not always applicable. This will become clear when we describe our algorithm in Section 3.2.4.

Finally, we will assume that we have access to quantum read/classical write RAM (QRAM), where a single QRAM operation is considered to be classically writing a bit to the QRAM or making a quantum query (a read operation) to bits stored in QRAM, possibly in superposition. See [4] for a more detailed discussion.

3.2 Quantum community detection

In the sections that follow we describe our various quantum algorithms for community detection. These algorithms are (roughly in order of increasing simplicity):

- **QLouvain** – A quantum version of classical Louvain (Section 3.2.1).
- **SimpleQLouvain** – A much simplified version of **QLouvain** that deviates slightly from the behaviour of the original Louvain algorithm (Section 3.2.2).
- **QLouvainSG** and **SimpleQLouvainSG** – Versions of both algorithms above that are more efficient if the input graph is sparse (Section 3.2.3).

- **EdgeQLouvain** and **NodeComQLouvain** – Two vastly simplified algorithms that deviate substantially from the spirit of the original Louvain algorithm, but nevertheless obtain similar results in practice (Section 3.2.5).

We also describe an approach based on variable time amplitude amplification in Section 3.2.4 that yields (asymptotically) more efficient versions of the first four algorithms above. However, these algorithms are much more complicated than those described above, and therefore we have chosen not to simulate these numerically in Section 5.

3.2.1 Quantum louvain

Our first quantum algorithm works by identifying the first vertex in a list for which there exists a good move, and then moves it, just as the classical algorithm does. Using the quantum algorithm **FindFirst** (introduced below) we obtain in this way a square-root improvement over the per-step classical complexity.

We begin by describing a quantum algorithm that performs a quantum search over a list of vertices in order to identify one for which a good move exists. The algorithm comes with a bound on the *expected* run-time – which benefits from having more good moves and good vertices available – and a bound on the *worst-case* run-time, which forgoes the aforementioned benefits. We will use the latter bound in our analysis of the main algorithm, since it is insensitive to the number of marked items, but in fact the run-time would be improved in practice by taking into account the actual number of good vertices.

Lemma 5 *There exists a quantum algorithm* **VertexFind**(L,ζ)*, which, given a list L of vertices $u_0,\dots,u_{|L|-1}$, returns the identity i of a vertex u_i such that $\bar{\Delta}_{u_i} > 0$ (i.e. a good vertex) with probability $\geq 1-\zeta$ if one exists, and otherwise returns 'no vertex exists'. The algorithm requires an expected number of function calls at most*

$$O\left(\sqrt{\frac{\delta_{\max}}{f}}\log\left(\frac{|L|}{\zeta}\right)\right) = \tilde{O}\left(\sqrt{\frac{\delta_{\max}}{f}}\log\left(\frac{1}{\zeta}\right)\right),$$

and

$$O\left(\sqrt{\frac{\delta_{\max}}{f}}\log(|L|)\log(\delta_{\max})\log\left(\frac{|L|}{\zeta}\right)\right) = \tilde{O}\left(\sqrt{\frac{\delta_{\max}}{f}}\log\left(\frac{1}{\zeta}\right)\right)$$

elementary operations, where f is the fraction of vertices in L that are good (and the $\tilde{O}$ notation hides polylogarithmic factors in $|L|$ and $\delta_{\max}$). If we want to obtain a worst case *run-time, then there is a variant of the algorithm that behaves the same, but requires in the worst case at most*

$$O\left(\sqrt{\delta_{\max}|L|}\log\left(\frac{|L|}{\zeta}\right)\right) = \tilde{O}\left(\sqrt{\delta_{\max}|L|}\log\left(\frac{1}{\zeta}\right)\right)$$

function calls and

$$O\left(\sqrt{\delta_{\max}|L|}\log(|L|)\log(\delta_{\max})\log\left(\frac{|L|}{\zeta}\right)\right) = \tilde{O}\left(\sqrt{\delta_{\max}|L|}\log\left(\frac{1}{\zeta}\right)\right)$$

elementary operations.

proof. We apply Grover search to find, for a particular vertex u, an integer $j \in [\delta_u]$ such that $\Delta_u^{\eta_u(j)} > 0$ (a 'good move'), if one exists. Using this as a subroutine, we apply Grover search now to the list L to find any vertex for which there exists such a neighbouring community (a 'good vertex').

Using the data structure described in Section 2.3.2, we can obtain the inputs to g_Δ, which computes the change in modularity resulting from moving a vertex u to a community α, in constant time: we can recover from η_u the quantity S_u^α (this is just the weight associated to the entry $\eta_u(j)$), and also obtain $S_u^{\ell(u)}$, s_u, Σ_a, and $\Sigma_{\ell(u)}$ directly in $O(1)$ time from the appropriate lists. We will use $\mathcal{A}_{g_\Delta}$ to denote the unitary that implements the (classical) sub-routine for computing $g_\Delta(s_u, \Sigma_\alpha, \Sigma_{\ell(u)}, S_u^\alpha, S_u^{\ell(u)}) =: \Delta_u^{\eta_u(j)}$ given u and j, and whose action on basis states is

$$|u\rangle\,|j\rangle\,|0\rangle \mapsto |u\rangle\,|j\rangle\,\left|\Delta_u^{\eta_u(j)}\right\rangle .$$

For a fixed vertex u, we can find a j such that $\Delta_u^{\eta_u(j)} > 0$ if one exists using the algorithm **QSearch** of Lemma 1, by providing $\mathcal{A}_{g_\Delta}$ as an oracle. The algorithm will require in the worst case $O(\sqrt{\delta_u}\log(1/\epsilon))$ uses of $\mathcal{A}_{g_\Delta}$ (and hence g_Δ) and its inverse, and $O(\sqrt{\delta_u}\log|L|\log(1/\epsilon))$ other operations to find one with probability at least $1-\epsilon$, or to signal that no such j exists as appropriate. We will write $\mathcal{A}_{\bar{\Delta}}$ to denote the unitary that implements this quantum algorithm for a given vertex u, i.e. it maps

$$|u\rangle\,|0\rangle \mapsto |u\rangle\,\left|\bar{\Delta}_u > 0?\right\rangle$$

where the last register on the right is 1 if $\bar{\Delta}_u = \max_j \Delta_u^{\eta_u(j)} > 0$, and 0 otherwise.

We then use $\mathcal{A}_{\bar{\Delta}}$ as a subroutine to search for a vertex u such that $\bar{\Delta}_u > 0$, i.e. for which there exists a j such that $\Delta_u^{\eta(j)} > 0$ (a good vertex). This can be achieved via another straightforward application of **QSearch** from Lemma 1, with $\mathcal{A}_{\bar{\Delta}}$ provided as the oracle. If the probability of a randomly chosen vertex u having a good move is $\frac{1}{f}$, $f > 0$, then this will require an expected $O(\sqrt{1/f})$ applications of $\mathcal{A}_{\bar{\Delta}}$ and its inverse, and when $f = 0$, the worst case, it will require at most $O(\sqrt{|L|})$ applications to signal that no there are no good vertices. Assuming that the sub-routine $\mathcal{A}_{\bar{\Delta}}$ works perfectly, we can boost the success probability of the algorithm from 2/3 to $1-\epsilon'$ by repeating $O(\log(1/\epsilon'))$ times. However, the subroutine $\mathcal{A}_{\bar{\Delta}}$ succeeds only with probability $\geq 1-\epsilon$. Hence for the outer search algorithm to work correctly we will need that every time the $\mathcal{A}_{\bar{\Delta}}$ subroutine is run, it succeeds. Since $\mathcal{A}_{\bar{\Delta}}$ (and its inverse) will be called at most $O(\sqrt{|L|}\log(1/\epsilon'))$ times, the entire search algorithm will therefore succeed with probability at least

$$(1-\epsilon')\cdot(1-\epsilon)^{O(\sqrt{|L|}\log(1/\epsilon'))}.$$

To ensure that this probability is $\geq 1-\zeta$, we can choose $1/\epsilon' = O(poly(1/\zeta))$ and $1/\epsilon = poly(|L|, 1/\epsilon')$. In particular, we can set $\epsilon' = \zeta/2$ and $\epsilon = \frac{\zeta/2}{\sqrt{|L|}\log(1/\epsilon')}$.

Finally, since the algorithm $\mathcal{A}_{\bar{\Delta}}$ is called in superposition for multiple vertices u, the run-time of the outer **QSearch** routine will be limited by its slowest branch,[g] which requires

[g]Note that the routine will really be limited by a known *upper bound* on the time taken by any particular branch. For us, this will be $O(\sqrt{\delta_{\max}}\log(1/\epsilon))$, but this does require us to know $\delta_{\max}$. Luckily, we can keep track of this over time by adding to our data structure, and the overheads for updating it will all at worst be logarithmic in n and linear in $d_{\max}$, similarly to the other updates.

at most $O(\sqrt{\delta_{\max}}\log(1/\epsilon))$ queries to g_Δ, plus $O(\sqrt{\delta_{\max}}\log(\delta_{\max})\log(1/\epsilon))$ other operations, meaning that the total expected number of function calls made by the entire algorithm is at most

$$O\left(\sqrt{\frac{\delta_{\max}}{f}}\log\left(\frac{|L|}{\zeta}\right)\right),$$

and the total expected number of other operations is

$$O\left(\sqrt{\frac{\delta_{\max}}{f}}\log(|L|)\log(\delta_{\max})\log\left(\frac{|L|}{\zeta}\right)\right).$$

Using the worst-case upper bound of **QSearch** from Lemma 1, we can bound the total *worst-case* number of function calls by

$$O\left(\sqrt{\delta_{\max}|L|}\log\left(\frac{|L|}{\zeta}\right)\right)$$

and the total number of other operations by

$$O\left(\sqrt{\delta_{\max}|L|}\log(|L|)\log(\delta_{\max})\log\left(\frac{|L|}{\zeta}\right)\right).$$

□

We will use the algorithm **VertexFind** as a subroutine to implement the quantum algorithm that searches an ordered list of vertices for the first vertex with a good move available. We can now describe this algorithm in detail.

Lemma 6 *Given an ordered list L of vertices $u_0,\dots,u_{|L|-1}$, there exists a quantum algorithm **FindFirst**(L,ϵ), which, with probability $\geq(1-\epsilon)$, returns $i=\min_j\{j:\bar{\Delta}_{u_j}>0\}$, i.e. the index of the first good vertex in the list if such a vertex exists, and otherwise returns 'no good vertex exists'. The algorithm requires at most*

$$O\left(\sqrt{\delta_{\max}i}\log\left(\frac{|L|}{\epsilon}\right)\right)$$

function calls and $\tilde{O}\left(\sqrt{\delta_{\max}i}\log\left(|L|/\epsilon\right)\right)$ other elementary operations in the case that there does exist a good vertex, and otherwise requires at most

$$O\left(\sqrt{\delta_{\max}|L|}\log\left(\frac{|L|}{\epsilon}\right)\right)$$

function calls and $\tilde{O}\left(\sqrt{\delta_{\max}|L|}\log\left(|L|/\epsilon\right)\right)$ other elementary operations.

proof. Let i be the index of the first good vertex in L, and let q be such that 2^q is the smallest power of 2 larger than i, and for clarity assume that the length of L is a power of 2 (this is without loss of generality – we can always pad L and incur at most a constant overhead in the run-time). The algorithm will proceed in two stages: first, we identify the segment of L in which i lies; then once we have identified the segment, we perform a binary search to identify the precise location of i in that segment.

Using the algorithm **VertexFind** from Lemma 5, we search over regions of L that double in size each time, in order to identify an upper bound j on i satisfying $i\leq j\leq 2^q$. In particular we repeat the following routine, initialising $l=0$ and $r=1$

1. Let $J = u_l, \ldots, u_r$ be the sub-list of vertices in L between l and r. Run **VertexFind**(J, ζ) to find a good vertex in J, or to determine (with probability $\geq 1 - \zeta$) that none exists. This requires at most $\tilde{O}(\sqrt{\delta_{\max}|J|}\log(1/\zeta))$ function calls and other elementary operations.

2. If a good vertex was found at index j, then return l and j and stop.

3. Otherwise set $l \leftarrow r + 1$ and $r \leftarrow 2r$. If $l > |L|$, return 'no good vertex exists' and stop; otherwise go to step 1.

If the above routine fails to find a good vertex, then we can simply output 'no good vertex exists' and stop. If instead there is a good vertex at position i, then with high probability (specifically $\geq (1-\zeta)^q$) we will detect this, and output an index j that we know to be an upper bound to i. It is only an upper bound since one segment of L might contain multiple good vertices and **VertexFind** will return any one of these, and so all we learn is that the vertex we're looking for is either at that position or before it. Similarly, since **VertexFind** will have failed to find any good vertex in any preceding segment that doesn't include i, we know a lower bound on the position of i, namely l.

From Lemma 5, the number of function calls made by **VertexFind** on a list of size a with failure probability ζ is $O\left(\sqrt{\delta_{\max}a}\log\left(\frac{a}{\zeta}\right)\right)$, and hence the run-time of the above procedure to find the segment of L containing i is

$$\begin{aligned}\sum_{k=0}^{q} O\left(\sqrt{\delta_{\max}2^k}\log\left(\frac{2^k}{\zeta}\right)\right) &\leq \sum_{k=0}^{q} O\left(\sqrt{\delta_{\max}2^k}\log\left(\frac{i}{\zeta}\right)\right) \\ &= O\left(\sqrt{\delta_{\max}2^q}\log\left(\frac{i}{\zeta}\right)\right) \\ &\leq O(\sqrt{\delta_{\max}i}\log(i/\zeta))\,,\end{aligned}$$

where the first and last inequalities follow since $q = \lceil \log_2 i \rceil$ (and hence $2^q \leq 2i$).

Once we have lower and upper bounds on the value of i, we can perform a binary search to find i precisely. The procedure is the following, initialising $r = j$:

1. Set $c = \lceil \frac{l+r}{2} \rceil$, and let $J = u_l, \ldots, u_c$ be the sub-list of L containing the left half of the vertices indexed between l and r. If $|J| = 1$, classically check whether the vertex it contains is good or not. If it is, return l, otherwise return $l + 1$.

2. Run **VertexFind**(J, ζ') to attempt to find a vertex in J, which requires at most $O\left(\sqrt{\delta_{\max}(c-l)}\log\left(\frac{(c-l)}{\zeta'}\right)\right)$ function calls and other elementary operations.

3. If a marked vertex is found at position $l \leq j \leq c$, then set $r \leftarrow j$. Otherwise, set $l \leftarrow c$. Repeat from step 1 above.

This procedure (which is just binary search on L with a quantum subroutine) will return the index of the left-most good vertex with probability $\geq (1-\zeta')^{\lceil \log(a) \rceil}$ where a is the size of the segment of L identified by the preceding routine.

Once again, we are running **VertexFind** (with failure probability now ζ') on lists that halve in size each time, starting with one that is of size $a/2$. Hence, the total run-time of this part of the algorithm is at most

$$\begin{aligned}\sum_{k=0}^{\lceil \log(a/2) \rceil} O\left(\sqrt{\delta_{\max} 2^k} \log\left(\frac{2^k}{\zeta'}\right)\right) &\leq \sum_{k=0}^{\lceil \log(a/2) \rceil} O\left(\sqrt{\delta_{\max} 2^k} \log\left(\frac{a}{\zeta'}\right)\right) \\ &= O\left(\sqrt{\delta_{\max} a} \log\left(\frac{a}{\zeta'}\right)\right).\end{aligned}$$

Finally, we note that the segment containing i is of size at most i, and hence we find that the run-time of this part of the algorithm is also at most

$$O\left(\sqrt{\delta_{\max} i} \log\left(\frac{i}{\zeta'}\right)\right).$$

It remains to choose the failure probabilities ζ and ζ'. We require that both parts of the algorithm succeed with probability $\geq \sqrt{(1-\epsilon)}$ each. In order to achieve this, we require for the first part that $(1-\zeta)^q \geq \sqrt{(1-\epsilon)}$, and for the second part that $(1-\zeta')^{\lceil \log(i) \rceil} = (1-\zeta')^q \geq \sqrt{(1-\epsilon)}$. Both conditions can be satisfied by choosing $\zeta = \zeta' = \Omega(\epsilon/q) = \Omega(\epsilon/\log(|L|))$, yielding our final run-time. □

Note that we made use of the worst-case complexities for **VertexFind** in the above analysis, in particular using the variant of the algorithm that is not faster even when there are more good vertices available, and hence it is likely that in practice the algorithm will be much faster.

Finally, we can use the algorithm **FindFirst** to construct a quantum version of the Louvain algorithm. Recall that the classical algorithm constructs a randomly ordered list of all vertices, locates the first good vertex in this list, and then moves it. Then it chooses the next good vertex, and so on, repeating this process until no good vertices are found in the remainder of the list. The corresponding quantum algorithm is precisely the same as the classical Louvain algorithm, except that the step in which the classical algorithm looks for the next good vertex in the list of vertices is replaced by a single call to the **FindFirst** algorithm of Lemma 6.

To see how long this algorithm takes, suppose the classical algorithm makes T moves, and that the kth move necessitated inspecting t_k vertices before finding one that could be moved. Then as we saw in Section 2.3.2 the classical algorithm will make at most

$$\sum_{k \in [T]} O(\delta_{\max} t_k),$$

calls to g_Δ, and use

$$\sum_{k \in [T]} \tilde{O}(\delta_{\max} t_k + d_{\max})$$

other operations.[h] We will also use this data structure in the quantum version of the algorithm. Finally, note that this run-time is somewhat pessimistic – the $\delta_{\max}$ could be replaced with

[h] Here we assumed that the algorithm makes use of the additional data structure described in Section 2.3.2, which in particular allows us to obtain the inputs to g_Δ in constant time, in exchange for a $O(d_{\max})$-time update step after moving a vertex.

the average number of neighbouring communities amongst all vertices inspected during the kth step, which in general should be smaller. The quantum algorithm cannot take advantage of this fact, however, and is really limited by $\delta_{\max}$.

Theorem 1 *There exists a quantum algorithm* ***QLouvain*** *that, with probability* $\geq 2/3$*, behaves identically to the Louvain algorithm and requires at most*

$$\sum_{k\in[T]} \tilde{O}(\sqrt{\delta_{\max} t_k})$$

calls to g_Δ *and*

$$\sum_{k\in[T]} \tilde{O}(\sqrt{\delta_{\max} t_k} + d_{\max})$$

other elementary operations.

proof. We replace the part of the classical algorithm that searches for the next good vertex with a single call to **FindFirst**. Once we identify a good vertex, we can use the **QMax** algorithm of Lemma 3 to obtain the *best* move available for that vertex, with probability at least $1-\epsilon$, using at most $O(\sqrt{\delta_{\max}}\log(1/\epsilon))$ function calls and $O(\sqrt{\delta_{\max}}\log(\delta_{\max})\log(1/\epsilon))$ other operations. After making the move, we have to update the data structure used by the **VertexFind** subroutine, which incurs a time-cost of $O(d_{\max})$.

To obtain the quantum run-time, we first need to choose settings for the failure probabilities of the **FindFirst** and maximum-finding subroutines. In particular, we require that all calls to both routines succeed with probability $\geq \sqrt{2/3}$ each, and so we can choose the failure probability of both to be $\epsilon = O(1/T) = O(1/\,poly(n))$. Hence, the quantum version of Louvain will require at most

$$\sum_{k\in[T]} O(\sqrt{\delta_{\max} t_k}\log(n))$$

calls to g_Δ and

$$\sum_{k\in[T]} \tilde{O}(\sqrt{\delta_{\max} t_k} + d_{\max})$$

other elementary operations, yielding a square-root worst-case improvement over the classical algorithm. Since the quantum algorithm mimics (with constant probability) the behaviour of the classical one, it will produce exactly the same output. □

3.2.2 Simple quantum louvain

Rather than finding the *first* marked item in a list, which requires repeated Grover searches using a bisection method, for a quantum computer it is more natural to simply find *any* marked item, which requires only a single application of Grover search. Motivated by this observation, in this section we describe a slightly different version of the Louvain algorithm that has a much simpler quantum analogue.

Concretely, the simpler algorithm (i) searches over the list of all vertices until it finds a good vertex by sampling vertices at (uniformly) random *with replacement*; (ii) once a good vertex is found, we move it and go back to (i). This process is repeated until no good vertices can be found. The quantum version of this algorithm, which we call **SimpleQLouvain**, is as follows:

1. Call **VertexFind**(L, δ) with L the list of all vertices in the graph, and δ to be determined. If there are any vertices in the graph for which a good move is available, this will return one at random. Otherwise, it will signal that none exist and we can end this phase of the algorithm.

2. Assuming that **VertexFind**(L, δ) returned a vertex, we use the **QMax** algorithm of Lemma 3 to obtain the *best* move available to that vertex.

3. We move the vertex, update the data structure, and then repeat from Step 1.

We note that this algorithm is subtly different to the original Louvain algorithm: here we search *with replacement* over the vertices of the graph, whereas the original algorithm searchers *without replacement* by fixing a randomly ordered list of vertices and sequentially searching through it. We verify numerically in Section 5 that this difference does not qualitatively change the behaviour of the algorithm in any significant way.

For a T-step run, in order for **SimpleQLouvain** to succeed (i.e. find a good vertex whenever one exists) with probability $\geq 2/3$, we require that $(1-\delta)^T \geq 2/3$, which can be achieved by setting $\delta = 1/O(T) = 1/O(poly(n))$. In that case, if there are an f_k fraction of good vertices in the graph after having already made $k-1$ moves, the expected number of function calls (and other operations) of **VertexFind** will be at most $\tilde{O}(\sqrt{\delta_{\max}/f_k})$ as per Lemma 5. Finally, we can choose the failure probability δ' of the quantum maximum-finding routine to be $\delta' = \delta$. Hence, the overall algorithm will require an expected number of at most

$$\sum_{k\in[T]} \tilde{O}\left(\sqrt{\frac{\delta_{\max}}{f_k}}\right)$$

calls to g_Δ and

$$\sum_{k\in[T]} \tilde{O}\left(\sqrt{\frac{\delta_{\max}}{f_k}} + d_{\max}\right)$$

other operations.

In contrast, the expected number of function calls made by the equivalent classical algorithm (the one that also searches with replacement) is

$$\sum_{k\in[T]} O\left(\frac{\delta_{\max}}{f_k}\right), \tag{6}$$

and so the quantum algorithm is asymptotically more efficient for any step of that algorithm (both in terms of calls to g_Δ and in terms of other operations). In general, the final stages of the algorithm will have $1/f_k = O(n)$, and in these steps the quantum speedup is quite large.

Finally, we note that if we do not make use of the data structure for the quantum algorithm, then the number of function calls remains the same, but the number of other operations becomes $\sum_{k\in[T]} \tilde{O}\left(\frac{\sqrt{\delta_{\max}}+d_{\max}}{\sqrt{f_k}} + d_{\max}\right)$.

In practice, the $\delta_{\max}$ appearing in the classical run-time in Eq. (6) is overly pessimistic: it will in fact be closer to the average number[i] of adjacent communities, δ_{avg}, since the algorithm visits vertices one by one, computing $\bar{\Delta}_u$ for each in time $O(\delta_u)$. On the other hand the quantum algorithm really is limited by the maximum number of adjacent communities due to our use of Grover search, and hence in practice could find itself being slower. We observe in Section 5 that in fact this is indeed the case, and so the 'worst-case' quantum speedup that we find via an asymptotic analysis often doesn't materialise in practice. In Section 3.2.4 we describe a more sophisticated quantum algorithm that makes use of the technique of variable time amplitude amplification to remove the dependency on $\delta_{\max}$, in an effort to overcome this limitation.

3.2.3 Trading a square-Root for a log factor for sparse Graphs

In the quantum algorithms described above, the run-time contains a log factor that could in practice be quite large. For example, the run-time of **VertexFind**(L,ζ), which finds a good vertex in a list L (or confirms that there aren't any) with probability at least $1-\zeta$ is $O\left(\sqrt{\frac{\delta_{\max}}{f}}\log\left(\frac{|L|}{\zeta}\right)\right)$, where f is the fraction of vertices in L that are good. The $\log(|L|)$ overhead arises because the quantum algorithm performs a Grover search over the vertices in L, using another Grover search as a subroutine. In order for the outer search to succeed with probability at least $1-\zeta$, the inner Grover search has to succeed with a much larger probability, namely $\approx 1-\frac{\zeta}{\sqrt{|L|}}$ (since the outer search will make in the worst-case $O(\sqrt{|L|})$ calls to the inner search routine). In practice this might be a substantial overhead, especially if $\log(|L|)$ is large relative to $\delta_{\max}$. In our algorithms, the list L will often be of size $\Theta(n)$, and hence for very sparse graphs, for example when $\delta_{\max} \leq d_{\max} = O(\log n)$, this overhead will be large enough to negate the square-root speedup that we obtain in terms of $\delta_{\max}$.

Hence, for such sparse graphs it will often make sense to replace the inner Grover search with a purely classical routine that succeeds with certainty. We will use the suffix '**SG**' to signify that the inner loop over the neighbouring communities is classical. In this case we can construct an alternative version of **VertexFind**, in which the time taken to find a good vertex in L with probability $\geq 1-\zeta$ is now $O\left(\frac{\delta_{\max}}{\sqrt{f}}\log\left(\frac{1}{\zeta}\right)\right)$. Using this variant of **VertexFind** we can then construct different versions of the above quantum algorithms that might perform better on sparse graphs. It is straightforward to check that the alternative run-times of these new algorithms designed for sparse graphs will be the following.

- **VertexFindSG**(L,ζ):
  - Expected number of function calls at most
$$O\left(\frac{\delta_{\max}}{\sqrt{f}}\log\left(\frac{1}{\zeta}\right)\right).$$
  - Worst-case number of function calls
$$O\left(\delta_{\max}\sqrt{|L|}\log\left(\frac{1}{\zeta}\right)\right).$$

[i] But not exactly: it is actually the average over subsets of vertices containing precisely one good vertex. If the good vertices have many adjacent communities then this average will be biased towards this. We discuss this in more detail in Section 3.2.4.

- **FindFirstSG**(L,ϵ):
  - Worst-case number of function calls
$$O\left(\delta_{\max}\sqrt{|L|}\log\left(\frac{\log(|L|)}{\epsilon}\right)\right).$$
- **QLouvainSG**:
  - Worst-case number of function calls, if classical Louvain makes T moves, with t_k vertices inspected during move k:
$$\sum_{k\in[T]} O\left(\delta_{\max}\sqrt{t_k}\log\left(T\log(n)\right)\right).$$
- **SimpleQLouvainSG**:
  - Expected number of function calls, if the algorithm makes T moves, with f_k the fraction of good vertices available during move k:
$$\sum_{k\in[T]} O\left(\frac{\delta_{\max}}{\sqrt{f_k}}\log\left(T\right)\right).$$

Hence, the $\log(n)$ factor present in **QLouvain** becomes a $\log\log(n)$ factor, although the overhead of $\log(T)$ is still present in the new versions of both that algorithm and of **SimpleQLouvain**.

3.2.4 Quantum Louvain via variable time amplitude amplification

An unsatisfactory element of the algorithms from the previous sections is that the subroutine that computes $\bar{\Delta}_u$ takes a different amount of time for each u, but the outer Grover search of **VertexFind** is limited by its slowest branch and hence its run-time depends on $\delta_{\max}$, in contrast to the classical algorithm whose run-time depends on a value closer to δ_{avg}. For many families of graphs (e.g. power-law graphs), this discrepancy could be significant – i.e. it might not be unlikely that $\sqrt{\delta_{\max}} > \delta_{\text{avg}}$. In this section we describe a more sophisticated quantum algorithm, **VertexFindVTAA**(L,ζ), that searches for good vertices from a list L, and whose run-time is sensitive to the fact that most vertices will not have a number of neighbouring communities close to the maximum. Similarly to the previous sections, we can then use this quantum algorithm as a subroutine to construct quantum algorithms for community detection.

Our main technical tool is the *variable time amplitude amplification* algorithm of Ambainis [3]. Using this as a subroutine, we show

Theorem 2 *Given a list L of vertices such that a fraction $f > 0$ of them are good, and the unitaries $\mathcal{A}_c$ and $\mathcal{A}_s$ defined in Eqs.* (C.1) *and* (C.2)*, we can use variable time amplitude amplification to construct a quantum algorithm* ***VertexFindVTAA****(L,ζ) that makes an expected*
$$O\left(\left(\delta_{\max}\log(\delta_{\max}) + \frac{t^q_{avg}}{\sqrt{f}}\log^{1.5}\delta_{\max}\right)\log(1/\zeta)\right)$$

calls to g_Δ*, where*

$$t^q_{\text{avg}} = \sqrt{\sum_{i=1}^{\delta_{\max}} p_i i^2},$$

and that returns the identity of a good vertex and the best move available to it with probability $\geq 1-\zeta$*. If there is no good vertex, the algorithm will signal this and requires at most*

$$O\left(\left(\delta_{\max}\log(\delta_{\max}) + t^q_{avg}\sqrt{|L|}\log^{1.5}\delta_{\max}\right)\log(1/\zeta)\right)$$

queries to do so.
We defer to Appendix C for details of the algorithm and the proof of the theorem.

Note that by using this version of the **VertexFind** algorithm, we lose the square-root improvement of the dependence on $\delta_{\max}$ that we obtained with the simpler Grover-based quantum algorithm. In exchange for the square-root speed up, the dependence on $\delta_{\max}$ is improved to a dependence on something closer to δ_{avg}. The reason for losing this speed up is because we exchanged a quantum search over the neighbours of each vertex with a classical, sequential one. One might wonder whether we could retain the square-root speedup for this part of the algorithm by replacing the classical algorithm $\mathcal{A} = \mathcal{A}_c \cdots \mathcal{A}_c\mathcal{A}_s$ with a quantum subroutine, say, $\mathcal{Q} = \mathcal{Q}_q \cdots \mathcal{Q}_q$. However, this does not seem possible since existing quantum algorithms for search with an unknown number of marked items, such as the Grover search of Lemma 1, cannot be separated into fixed 'steps' $\mathcal{Q}_q$ that satisfy the conditions from [2] (and described in the proof of the theorem above) without assuming, for example, that every vertex has the same number of good moves – that is, to use VTAA with a subroutine $\mathcal{Q}$, the algorithm $\mathcal{Q}$ cannot be *adaptive*, and must instead act identically on every branch of the superposition to which it is applied (in our case, on each vertex $u \in V$). It is an interesting open question whether the techniques from [2] can be extended to such an adaptive setting, particularly in the case where the subroutine used in VTAA is a quantum search over an unknown number of marked items.

Finally, we remark that, due to its very complex nature, it was extremely difficult to precisely pin down the exact number (including constants) of function calls required by the VTAA-based quantum algorithm.[j] For this reason we did not numerically simulate the algorithm in order to compare it to our other quantum (and classical) algorithms for community detection.

3.2.5 Quantum Louvain algorithms utilizing different search-spaces

In this section we consider a vastly simplified algorithm, called **EdgeQLouvain** which only utilizes a single Grover search over a large search space, rather than searching over multiple search spaces (vertices and their neighbours).[k] The algorithm differs somewhat in spirit to the original Louvain algorithm described in Section 2.3, however as we show in Section 5, tends to yield similar results in terms of the modularity it obtains.

[j] In fact, we suspect that it is not possible to do so (without substantial work) from the description given in [2] alone.
[k] We also considered a similar algorithm that instead searches over (vertex, neighbouring-community) pairs. This algorithm gave very similar results to **EQL**, and therefore we restrict our attention only on **EQL**.

EdgeQLouvain, or **EQL** for short, searches over directed edges (u, v) of the graph for one that gives an increase in modularity if u is moved to community $\ell(v)$. Upon finding one, it moves u greedily to a neighboring community (but not necessarily to $\ell(v)$ itself). This approach has the two advantages that, in the quantum case, the Grover search does not need to make use of another nested Grover search over neighboring communities, and likewise there will now be only an additive dependency on either one of $\delta_{\max}$ or δ_{avg}, nullifying the two issues that we have encountered with our algorithms thus far.

Given the edge set E of the input graph for the Louvain algorithm, let $E_d = \{(u, v) : \{u, v\} \in E\}$ be the set of directed edges obtained by replacing every undirected edge $\{u, v\} \in E$ by both (u, v) and (v, u), making $|E_d| = 2|E|$. As usual, we assume that we have access to the data structure described in Section 2.3.2. The three phases of the algorithm operate as follows:

Initialization – Exactly the same initialization procedure as in OL (original Louvain), see Section 2.2 for details.

First phase – Use **QSearch** to search over all edges (u, v) in search of one that yields a good move. As an oracle we provide the unitary that, for a pair (u, v), computes whether $\Delta_u^{\ell(v)}$ is positive or not, which can be done using $O(1)$ function calls to g_Δ and $O(\log \delta_u)$ other operations (to obtain the inputs to g_Δ we need to perform a binary search over the community adjacency list of u to find the entry corresponding to the community of v, whilst the other inputs can be obtained in constant time).

Instead of moving u to $\ell(v)$, we find the *best* neighbouring community of u, $\bar{\alpha} = \arg\max_{\alpha \in \zeta_u} \Delta_u^\alpha$, by using quantum maximum finding over all moves to neighbouring communities of u. We then update the data structure as we do for OL in time $\sum_{v \in N_u} O(\log \delta_v)$ (see Section 2.3).

Second phase – Again this is identical to the second phase of OL; see Section 2.2.

By using **QSearch**, we can find a pair (u, v) yielding a good move with an expected

$$O\left(\frac{1}{\sqrt{h_k}} \log(1/\delta)\right)$$

number of calls to g_Δ, and

$$\tilde{O}\left(\frac{1}{\sqrt{h_k}} \log(1/\delta)\right)$$

other operations, where h_k is the fraction of edges that yield a good move during the kth step. If no such pair exists then the algorithm will signal this after making at most $O(\sqrt{|E|} \log(1/\delta))$ queries to g_Δ.

Once we have found such a vertex, with probability $\geq 1 - \epsilon$ we can find the best move available using the **QMax** algorithm of Lemma 3, which will require at most $O(\sqrt{\delta_{\max}} \log(1/\epsilon))$ calls to g_Δ and $O(\sqrt{\delta_{\max}} \log(\delta_{\max}) \log(1/\epsilon))$ other operations. If the algorithm makes T moves in total, we will need to choose ϵ and δ such that all calls to either **QSearch** or quantum maximum-finding will succeed with sufficiently high likelihood that the probability that any one of them fails is at most $2/3$, which can be satisfied by choosing $\epsilon = \delta = 1/O(T) =$

$1/O(poly\, n)$. Hence, the algorithm will make an expected number of function calls at most

$$\sum_{k\in[T]} \tilde{O}\left(\frac{1}{\sqrt{h_k}} + \sqrt{\delta_{\max}}\right)$$

and

$$\sum_{k\in[T]} \tilde{O}\left(\frac{1}{\sqrt{h_k}} + \sqrt{\delta_{\max}} + d_{\max}\right)$$

other operations.

The complexity of this algorithm appears to be favourable compared to the algorithms from the previous sections, and has the additional advantage of being very simple and therefore incurring smaller logarithmic overheads. As we show in Section 5, it also behaves similarly to the original Louvain algorithm in practice, whilst being the fastest amongst all quantum algorithms that we evaluated.

4 Estimating the run-times of quantum algorithms for community detection

In this section and the next we use the tools and methodology of [9] to empirically estimate the run-times (more precisely the number of queries to the (gradient of) the modularity function) of our quantum algorithms for a variety of inputs, and use these estimates to compare their performances, to each other and to their classical counterparts. This allows us to estimate how much of the per-step speedup suggested by the asymptotic analyses in Section 3 manifests in the final behaviour of the algorithms, for a range of inputs. We find for all algorithms that some speedup does make it out, though to varying degrees. Moreover we observe that the algorithms that promise the greatest speedups through an asymptotic analysis are not necessarily the ones that achieve the best speedups 'in practice'. In our view, this demonstrates the usefulness of this sort of analysis over a purely asymptotic, worst-case one for designing efficient quantum algorithms to use for practical tasks.

In the sections that follow we describe explicitly our approach to simulating our quantum algorithms and estimating their expected run-times. We will focus on **QLouvain**, **Simple-QLouvain** and **EdgeQLouvain** (including both their original and sparse-graph versions), all of which fit into the framework of Algorithm 1 introduced in Section 1. We deliberately chose to forgo simulating the algorithms that make use of variable time amplitude amplification (VTAA) as a subroutine, not only because the nature of VTAA makes it difficult to do so, but also because the expected speedup will only be a constant given that the Louvain algorithm is predominantly applied to sparse graphs.

4.1 Complexity bounds

The first step is to obtain tight bounds (including all constants etc.) on the complexities of the quantum sub-routines that we make use of. We begin by recalling the complexity bounds obtained in [9] for the two quantum sub-routines that we use here: Grover search with an unknown number of marked items (**QSearch**, Lemma 1), and quantum maximum-finding (**QMax**, Lemma 3).

Expected complexity of Grover search As we mentioned in Section 2, when considering the full run-time, including constants, of **QSearch**, there is an extra hyper-parameter N_{samples}

used to determine the number of classical samples that are drawn before Grover search is used. Then the worst-case expected complexity of **QSearch** is as follows

Lemma 7 (Worst-case expected complexity of QSearch, [Lemma 4, [9]]) *Let L be a list, $g : L \to \{0,1\}$ a Boolean function, $N_{samples}$ a non-negative integer and $\epsilon > 0$, and write $t = |g^{-1}(1)|$ for the (unknown) number of marked items of L. Then, $\boldsymbol{QSearch}(L, N_{samples}, \epsilon)$ finds and returns an item $x \in L$ such that $g(x) = 1$ with probability at least $1-\epsilon$ if one exists using an expected number of queries to g that is given by*

$$E_{\boldsymbol{QSearch}}(|L|, t, N_{samples}, \epsilon) = \frac{|L|}{t}\left(1 - \left(1 - \frac{t}{|L|}\right)^{N_{samples}}\right) + \left(1 - \frac{t}{|L|}\right)^{N_{samples}} c_q E_{Grover}(|L|, t)\,, \tag{7}$$

where

$$E_{Grover}(|L|, t) \leq F(|L|, t)\left(1 + \frac{1}{1 - \frac{F(|L|,t)}{\alpha\sqrt{|L|}}}\right)\,, \tag{8}$$

with

$$F(|L|, t) = \begin{cases} \frac{9}{4}\frac{|L|}{\sqrt{(|L|-t)t}} + \left\lceil \log_{\frac{6}{5}}\left(\frac{|L|}{2\sqrt{(|L|-t)t}}\right)\right\rceil - 3 \leq \frac{\alpha\sqrt{L|}}{3\sqrt{t}} & \text{for} \quad 1 \leq t < \frac{|L|}{4} \\ 2.0344 & \text{for} \quad \frac{|L|}{4} \leq t \leq |L|. \end{cases} \tag{9}$$

If no marked item exists, then the expected number of queries to g equals the number of queries needed in the worst case (denoted by $W_{\boldsymbol{QSearch}}(|L|, N_{samples}, \epsilon)$), which is given by

$$E_{\boldsymbol{QSearch}}(|L|, 0, N_{samples}, \epsilon) = W_{\boldsymbol{QSearch}}(|L|, N_{samples}, \epsilon) \leq N_{samples} + \alpha c_q \lceil \log_3(1/\epsilon)\rceil)\sqrt{|L|}\,. \tag{10}$$

In the formulas above, c_q is the number of queries to g required to implement the oracle $\mathcal{O}_g |x\rangle |0\rangle = |x\rangle |g(x)\rangle$, and $\alpha = 9.2$.

Worst-case complexity of Grover search Using a modified version of **QSearch** (which we call $\mathbf{QSearch}_{\text{Zalka}}$, described in [9] and based on the algorithm in [33]), we can obtain an algorithm with better complexity in the case of no marked items.

Lemma 8 (worst-case complexity of $\mathbf{QSearch}_{\mathbf{Zalka}}$, [Lemma 5, [9]]) *Let L be a list of items, $g : L \to \{0,1\}$ a Boolean function and $\epsilon > 0$, and write c_q for the number of queries to g required to implement the oracle $\mathcal{O}_g |x\rangle |0\rangle = |x\rangle |g(x)\rangle$. Then, with probability of failure at most ϵ, $\boldsymbol{QSearch}_{Zalka}$ requires at most*

$$W_{\boldsymbol{QSearch}_{Zalka}}(|L|, \epsilon) := c_q\left(5\left\lceil \frac{\ln(1/\epsilon)}{2\ln(4/3)}\right\rceil + \pi\sqrt{|L|}\sqrt{\left\lceil \frac{\ln(1/\epsilon)}{2\ln(4/3)}\right\rceil}\right) \tag{11}$$

queries to g to find a marked item of L, or otherwise to report that there is none.

Quantum maximum-finding For maximum finding, we have the following result from [9].

Lemma 9 (Expected complexity of QMax, [Corollary 1, [9]]) *Let L be a list of items of length $|L|$ and $R : L \to \mathbb{R}$ a function that assigns a value to each item. Let f_i be the marking function*

$$f_i(j) = \begin{cases} 1 & \text{if} \quad R(j) > R(i) \\ 0 & \text{otherwise}. \end{cases}$$

*Then the expected number of queries to f_i (for any i) required for **QMax** to find the maximum of L with success probability at least $1-\epsilon$ is $\lceil \log_3(1/\epsilon) \rceil 3E_{\boldsymbol{QMax}_\infty}(|L|)$, where*

$$E_{\boldsymbol{QMax}_\infty}(|L|) \leq c_q \sum_{t=1}^{|L|-1} \frac{F(|L|,t)}{t+1}\,, \tag{12}$$

with $F(|L|,t)$ given by (9), and where c_q is the number of queries to f_i required to implement oracle access to f_i (which we assume to be the same for all i).

Using these bounds, we proceed to obtain bounds on the expected complexities of the main quantum subroutines used by our quantum community detection algorithms – **VertexFind** and **FindFirst**. Afterwards, in Section 4.2 we describe precisely how we simulate our quantum community-detection algorithms, including what we implement classically and what information we gather along the way, as well as what accuracy and hyperparameter settings we use.

*4.1.1 Expected complexity of **VertexFind***

To start with, we bound the expected complexity of the **VertexFind** algorithm when it is run on a list L containing $|L|$ vertices, t of which are good vertices, and when it is required to fail with probability at most ζ. Following the analysis in the proof of Lemma 5, we run **QSearch**(L,N_{samples},ϵ), giving it access to a subroutine **QSearch**$_{\text{Zalka}}$ (J,ϵ'), where J will be a list of length at most $\delta_{\max}$, and ϵ and ϵ' are parameters that are determined by ζ and (in the case of ϵ') the worst-case complexity of the outer **QSearch**(L,N_{samples},ϵ) routine. Note that, in order to implement the oracle required for Grover search, the subroutine **QSearch**$_{\text{Zalka}}$ (J,ϵ') and its inverse will each be run once per query.

On a list L of size $|L|$ and with failure probability at most ϵ, the *outer* **QSearch** routine that we use requires in the worst case (i.e. when $t = 0$) at most $W_{\mathbf{QSearch}}(|L|, N_{\text{samples}}, \epsilon)$ queries to its oracle/subroutine (see Eq. (10)). When there are $t > 0$ marked items, it requires an expected $E_{\mathbf{QSearch}}(|L|, t, N_{\text{samples}}, \epsilon)$ queries (see Eq. (7)). The *inner* **QSearch**$_{\text{Zalka}}$ routine is slightly different, and it requires in the worst case at most $W_{\mathbf{QSearch}_{\text{Zalka}}}(|J|, \epsilon')$ queries on a list J of size $|J|$ and with failure probability at most ϵ' (see Eq. (11)).

Finally, we need to set the failure probabilities appropriately to align them with the overall failure probability ζ for **VertexFind**. From Lemma 5 we find that to achieve a success probability $\geq 1-\zeta$, we can set $\epsilon = \zeta/2$ and $\epsilon' = \frac{\zeta}{2W_{\mathbf{QSearch}}(|L|, N_{\text{samples}}, \epsilon)}$. Putting everything

together, the expected complexity of the entire **VertexFind** algorithm will be at most

$$\begin{aligned} &E_{\mathbf{VertexFind}}(|L|, t, N_{\text{samples}}, \zeta) \\ &\quad = E_{\mathbf{QSearch}}(|L|, t, N_{\text{samples}}, \zeta/2) \cdot 2W_{\mathbf{QSearch}_{\text{Zalka}}}\left(\delta_{\max}, \tfrac{\zeta}{2W_{\mathbf{QSearch}}(|L|, N_{\text{samples}}, \zeta/2)}\right) \\ &\quad \leq E_{\mathbf{QSearch}}(|L|, t, N_{\text{samples}}, \zeta/2) \cdot 2c_q \Bigg[5 \left\lceil \ln\left(\tfrac{2W_{\mathbf{QSearch}}(|L|, N_{\text{samples}}, \zeta/2)}{\zeta}\right) / (2\ln(4/3)) \right\rceil \\ &\qquad\qquad + \pi\sqrt{\delta_{\max}} \sqrt{\left\lceil \ln\left(\tfrac{2W_{\mathbf{QSearch}}(|L|, N_{\text{samples}}, \zeta/2)}{\zeta}\right) / (2\ln(4/3)) \right\rceil} \Bigg], \end{aligned} \tag{13}$$

where the expression for $E_{\mathbf{QSearch}}$ can be found in Eq. (7) and the expression for $W_{\mathbf{QSearch}}$ in Eq. (10).

*4.1.2 Expected complexity of **VertexFindSG***

Next, we bound the expected complexity of the sparse-graph version of **VertexFind** (described in Section 3.2.3) when it is run on a list L containing $|L|$ vertices, t of which are good, and when it is required to fail with probability at most ζ. This time the inner $\mathbf{QSearch}_{\text{Zalka}}$ routine is eliminated, and hence the complexity of this algorithm depends only on a single application of **QSearch**, given access to a classical sub-routine that makes $\delta_{\max}$ queries per call. Similarly to the above case, this classical algorithm must be run twice in order to implement the oracle required for the **QSearch** routine. The expected complexity of the entire **VertexFind** algorithm will now be at most

$$E_{\mathbf{VertexFindSG}}(|L|, t, N_{\text{samples}}, \zeta) \quad = \quad E_{\mathbf{QSearch}}(|L|, t, N_{\text{samples}}, \zeta) \cdot 2\delta_{\max}\,. \tag{14}$$

*4.1.3 Expected complexity of **FindFirst** and **FindFirstSG***

The **FindFirst** routine makes a number of repeated calls to **VertexFind**, whose expected complexity on a list L with t marked items, and failure probability at most ζ, is given by $E_{\mathbf{VertexFind}}(|L|, t, N_{\text{samples}}, \zeta)$ from Eq. (13). In order to choose the correct setting of ζ to ensure that **FindFirst** fails with probability at most μ, we need to know the maximum number of times that **VertexFind** might be called by **FindFirst**. The worst case is when there is a single marked item lying at the very end of the list L. In this case, **FindFirst** will make $\lceil \log_2 |L| \rceil$ calls to **VertexFind**, on sets of increasing size, followed by a binary search that will require another $\lceil \log_2 \frac{|L|}{2} \rceil$ calls. In total **FindFirst** will call **VertexFind** at most $2\lceil \log_2 |L| \rceil - 1$ times, and hence to ensure that it fails with probability $\leq \mu$ we must ensure that every call to **VertexFind** is made with failure probability at most $\zeta \leq \frac{\mu}{2\lceil \log_2 |L| \rceil}$. Precisely the same analysis holds for **FindFirstSG**, which will make calls instead to **VertexFindSG**, again with failure probability at most $\zeta \leq \frac{\mu}{2\lceil \log_2 |L| \rceil}$.

4.2 Simulation details

In order to simulate any of our quantum community detection algorithms, we run a corresponding classical version of the algorithm of interest, and collect the information necessary to estimate how long the quantum algorithm would have taken in expectation. The classical algorithms we use are all based on an implementation of Louvain in Python by Aynaud [5].

During Phase 1 of (any version of) the Louvain algorithm (the part that is most time consuming and also benefits from a quantum speedup), we obtain our estimates for the complexities of our quantum algorithms by applying the upper bounds on the expected complexities of the **QSearch** and **VertexFind** algorithms given in Section 4.1 above. Our upper bounds depend on four parameters: the size $|L|$ of the list L to which the subroutines are applied, the desired failure probability ζ of the algorithm, the number t of marked vertices in the list, and the choice of hyper-parameter N_{samples}. In the sections below, we discuss how to obtain values for these parameters. In particular:

- The sizes $|L|$ of the lists will be available during the course of the simulation.
- The number t of marked vertices, however, is not immediately accessible – we discuss below in Section 4.2.1 how to obtain this value during the execution of the classical Louvain algorithm.
- The failure probabilities ζ for each sub-routine are determined by the overall acceptable failure probability for the entire algorithm, which requires knowing how many times each subroutine will be called. Of course, this information is not known ahead of time, and we discuss this in Section 4.2.3.
- N_{samples} is a hyper-parameter that changes the efficiency of the algorithms. Its optimal setting is discussed in Section 4.2.4

Finally, in order to be able to estimate the number of queries used by the different quantum algorithms introduced in Section 3, the corresponding implementations of the classical Louvain algorithm differ in places to that of Aynaud's. The differences in each case concern how precisely we find a vertex to move to a new community, and we discuss our implementations in Section 4.2.2.

4.2.1 Computing the number of marked items

Whenever we want to know how many queries a call to **QSearch** uses, we need to know the number of marked items. Below we describe our methods for obtaining this value while we simulate the quantum algorithm.

For the Louvain algorithm, the number of marked items can be kept track of exactly with the help of an additional data structure in the form of a set. We call this set $V_{\text{marked}} = \{u \in V | \exists \alpha \text{ such that } \Delta_u^\alpha > 0\}$, the set of marked vertices, i.e. those that have a neighbouring community they can move to in order to increase the modularity. With this set we can easily compute $t = |V_{\text{marked}}|$. The simplest way to obtain V_{marked} is by performing an exhaustive search over the entire list of vertices and explicitly checking for every vertex if it is marked or not (i.e. has a good move). This exhaustive search should be repeated after every move, since a move will change which vertices are marked and which are not. An exhaustive search after every move is tractable for small graphs, but quickly becomes intractable as the number of vertices increases.

Instead of recreating the list V_{marked} from scratch after every move, it can be initialized at the start and then updated after every move. This can be done by searching through the set of vertices that a particular move could possibly affect. More precisely, let u be the vertex that was moved from community α to community β in the previous step. The only vertices

that can be flipped from marked to non-marked and vise versa are those that belong to either community α or β or are neighbours of these communities. More specifically, the vertices contained in $V_{\text{changeable}} = N_{C_\alpha} \cup N_{C_\beta}$, where $N_{C_\alpha} = \{w \in N_v | v \in C_\alpha\}$ and likewise for N_{C_β}. This set of vertices is relatively small compared to the set of all nodes $|V| >> |V_{\text{changeable}}|$, which gives a fast update procedure for V_{marked}. After every move we construct the set $V_{\text{changeable}}$ as described above. For all vertices in $V_{\text{changeable}}$ we calculate Δ_u. If $\Delta_u > 0$, and if u is not in V_{marked}, we add u to V_{marked}. Similarly, if $\Delta_u \leq 0$, and u is in V_{marked}, we remove u from V_{marked}.

The added benefit of creating and maintaining this list, besides explicitly knowing t, is that we can directly sample from it to get a marked vertex. This in fact speeds up the search process of Louvain tremendously.[l] At every move we can now use $t = |V_{\text{marked}}|$ as an input to the bounds for the number of queries that our quantum algorithms would have made. This method is especially fast when the input graphs are sparse.

When searching over edges, as is done in **EdgeQLouvain**, we use the same procedure, but for marked edges rather than vertices.

It should be noted that, as discussed in [9], there are also sampling methods to estimate the number of marked items. We found that in the particular case of the Louvain algorithm, the sampling methods actually were not more efficient that simply keeping track of the aforementioned data structure, and therefore we have not included our results obtained through sampling in this paper. We do find that the number of function calls obtained using either the exact method described above or the sampling method described in [9] give total query counts that are quantitatively the same.

4.2.2 Classical simulation of the various Louvain algorithms

The program that simulates **QLouvain** runs the original Louvain with the addition of a subroutine to keep track of the number of queries. This subroutine is called whenever the first marked vertex is found. It estimates the number of function calls that **QLouvain** would have made to find this vertex. To do so, it simulates the behaviour of **FindFirst** ,as explained in 4.2.2, by searching over lists with varying sizes. For every list L with length $|L|$ smaller than a certain $|L|_{switch}$, a hyper parameter described in Section 4.2.4, **FindFirst** uses classical search to find the marked item, else it uses **VertexFind**. The parameters that are needed to calculate the number queries that **VertexFind** makes are set as follows: $|L|$ is the size of the respective list, ζ (the precision) is set as described in Section 4.2.3, and t is calculated directly from the list by the use of an exhaustive search. Adding these estimates together gives an estimate of the queries that **FindFirst** uses per move. The total number of queries is the sum of queries made at every move. During a single run we calculate the number of queries made by both **VertexFind** and **VertexFindSG**, using the same parameters, so that we get an estimate for both **QLouvain** and **QLouvainSG** in one pass.

SimpleQLouvain uses the subroutine **VertexFind** on the set of all vertices to find a random marked one. For this we can use our method described in Section 4.2.1 to obtain a marked vertex *and* an explicit calculation of t. After every move we compute the number of function calls that **VertexFind** would have made by using the following parameters:

[l] One could think that this would be a faster method for classical Louvain as search is done in a constant number of steps. It turns out that this is not the case, as updating the data structure still requires a lot of steps.

$|L| = |V|$ the number of vertices in the graph, ζ (the precision) is set as described in Section 4.2.3. In the same simulation we also calculate upper bounds for **VertexFindSG**, using the same parameters, so that again we get an estimate of the number of function calls for both **SimpleQLouvain** and **SimpleQLouvainSG** in a single pass.

EdgeQLouvain uses the subroutine **Qsearch** on the set of edges to find a marked edge. Similar to **SimpleQLouvain**, this can be simulated by keeping a list of all marked edges, as described in Section 4.2.1. After every move we estimate the number of queries **Qsearch** would have made using the parameters: $L = 2|E|$ ($|L| = 2|V|\delta_{\text{avg}}$), because the search is over directed edges as described in Section 3.2.5, ζ (the precision) is set as described in 4.2.3, and t is obtained using the methods described in Section 4.2.1.

Finally, once we have identified a good vertex u found by either **VertexFind**, **FindFirst** or **QSearch** (in case of **EdgeQLouvain**), we have to determine to what neighboring community u should be moved by computing $\arg\max_{\alpha \in \zeta_u}(\Delta_u^\alpha)$. We can find the maximum either classically, or with the quantum maximum-finding subroutine of Lemma 3. Since we keep track of the neighbouring communities in our data-structure, we known δ_u, and therefore we can decide beforehand whether, in expectation, it would be faster to either perform classical maximum-finding, or quantum maximum-finding. For the sparse graphs simulated in the results section, it turns out that classical maximum-finding always uses fewer queries, and quantum maximum-finding is therefore never used.

4.2.3 Setting the failure probabilities of the subroutines

One piece of information that we will not be able to obtain without running the entire algorithm first is the total number of moves that the algorithm will make, which determines the maximum number of times each subroutine might be called, which in turn determines the maximum acceptable failure probability for those subroutines. Of course, this is not a problem unique to our use-case, but to any speedup of such a heuristic algorithm. One option is to use the trivial upper bound from Appendix A.1 of $O(poly(n))$ total moves, however this is almost certainly a huge over-estimate, and in fact in practice it is observed that the Louvain algorithm makes only about $O(n \log n)$ moves before stopping [20]. In Appendix A.2 we present data that further justifies using such an estimated bound on the number of moves.

The only practical way to deal with this issue is to decide beforehand on an upper bound M to the number of moves T, perhaps based on empirical observations, and then use this to set the failure probabilities of the various subroutines. If the actual number of moves goes beyond M, then we conclude that we can no longer guarantee that the algorithm ran successfully. However, since the algorithm is anyway a heuristic, and the success of the algorithm is determined on more of a qualitative level than a quantitative one, some small number of failures (say, in choosing the 'wrong' vertex to move, or not finding the true maximum amongst all possible moves for a single vertex) may be entirely acceptable. In Section 5 we give empirically determined estimates for M based on the size of the input graph, and then use these to derive sensible failure probabilities for the main quantum subroutines used for each algorithm. These then determine the failure probabilities for the other subroutines, as described in Section 4.1.

4.2.4 Hyper-parameter settings

Optimal number of N_{samples} As discussed, the number of classical samples used in the implementation of **QSearch** is a hyper-parameter that can be tuned to optimize the run-time depending on the size of the list L, and what fraction f of the items in L are marked. Classical sampling requires fewer queries than Grover search does when a large fraction of the items is marked, whereas it is more efficient to not use classical sampling at all when a small number of them is marked. In [9], it was found that (with f_0 being the value of the fraction for which the expected number of queries for Grover search and classical sampling are equal)

- For $|L| \leq 260$, classical sampling always requires fewer queries.
- For $|L| \geq 260$, the point $1/f_0$ for which Grover search becomes more efficient than classical sampling is plotted as a function of $|L|$ in Fig. 1.

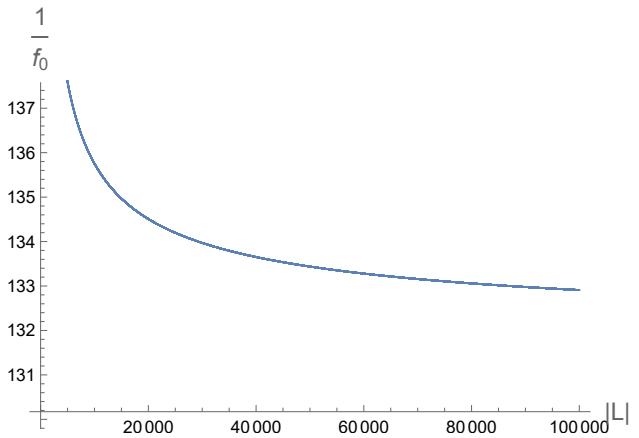

Fig. 1. The value of $1/f_0$ as a function of the list length $|L|$ that marks the point beyond which, in expectation, Grover search requires fewer queries than sampling classically does. In the limit $|L| \to \infty$, there is a horizontal asymptote at $1/f_0 \to 131.665$.

For (any variant of) the Louvain algorithm, we know from numerical results that the number of good vertices roughly decreases monotonically during a single iteration of the first phase of the algorithm. We can use this knowledge to set a criterion for $1/f$ beyond which it becomes more efficient to skip the classical sampling step all together and set $N_{\text{samples}} = 0$. Since the number of marked items decreases monotonically only approximately, we allow ourselves some wiggle room. As a consequence, in our simulations, we have chosen to use the following settings[m]

- We set $N_{\text{samples}} = 130$ at the start of phase 1. (An extra analyses of [9] suggests setting N_{samples} slightly lower than $\frac{1}{f_0}$)
- The moment we draw 130 consecutive samples without finding a marked item (implying $1/f \gtrapprox 130$), we set $N_{\text{samples}} = 0$ and use only Grover search from this point on.

[m]The plots in the results section are insensitive to fine-tuning N_{samples} beyond the point that we have done.

Optimizing the number of classical samples in FindFirst **FindFirst** searches through sets of varying sizes to detect if they contain a marked element. If the size of the set is sufficiently small it is more efficient to classically search through the set (from start to finish) than to use the quantum **VertexFind** subroutine. This introduces another hyper-parameter $|L|_{\text{switch}}$ that determines when to switch from using classical search to **VertexFind**, depending on the list size. In our numerical results we found that classical search was in fact always faster than using **VertexFind** for the graph sizes we studied. This made it impossible to choose a good setting for this hyper-parameter.

To allow for a comparison to be made between classically traversing the list and using **VertexFind** within **FindFirst**, we choose to set $|L|_{\text{switch}}$ to a finite number: $|L|_{\text{switch}} = 512$. I.e., sets with less than 512 vertices were searched through classically and for sets with more items than 512 vertices we used **VertexFind**. The inner loop of **QLouvain**, searching through the neighbouring communities of a node, was always performed using **Qsearch** even when we searched through the set classically.

5 Numerical results

For all numerical results in this section we assume that we require the failure probability of the entire quantum algorithm to be a small constant (10^{-5}). We assume that the algorithms make no more than $n\log(n) =: M$ moves (see Section 4.2.3 and Appendix A.2 for justifications for choosing this number of moves), and that the algorithm fails if any subroutine fails within any of these moves. Hence, we set the failure probabilities ϵ of the main (i.e. outermost) quantum subroutines for each algorithm to

$$\left(10^{-5}\right)^{\frac{1}{M}} \geq \frac{10^{-5}}{M} = \frac{10^{-5}}{n\log(n)} =: \epsilon. \tag{15}$$

Finally, we always use the additional data-structure introduced in Sec. 4.2.1 to keep track of the number of good vertices exactly.

5.1 Artificial data-sets

Since we are interested in the run0time scaling as well as the absolute query counts of the classical and quantum Louvain algorithms, we introduce in this section two methods for generating benchmark networks of arbitrary size. One method relies on the well-known LFR method [22], in which node degrees and community sizes are sampled according to power law distributions[n] with exponents τ_1 and τ_2, respectively. The other input parameters are the total number of nodes n, the average degree $\langle d\rangle$ (or alternatively one can set the minimum degree $d_{\min}$), the maximum degree $d_{\max}$ (by default set to n), minimum and maximum community sizes $S_{\min}$ and $S_{\max}$ (by default set to $d_{\min}$ and $d_{\max}$, respectively) and a mixing parameter μ which specifies the fraction of neighbours of a node that do not belong to the node's own community. We will refer to graphs of this type as 'LFR-type graphs' and use the *NetworkX* implementation [17] as a network generator.

In the second graph generation method, which we will call 'FCS-type graphs', we fix the community size S and adopt an algorithm similar to that described in Ref. [30]. The graph parameters are now n, $\langle d\rangle$ and S, and the edges are drawn uniformly at random from all

[n] Distributions not uncommon for real-world networks.

possible edges. A description of our algorithmic implementation (which runs in time $\mathcal{O}(|E|)$) for FCS-type graph generation is given in Appendix D.

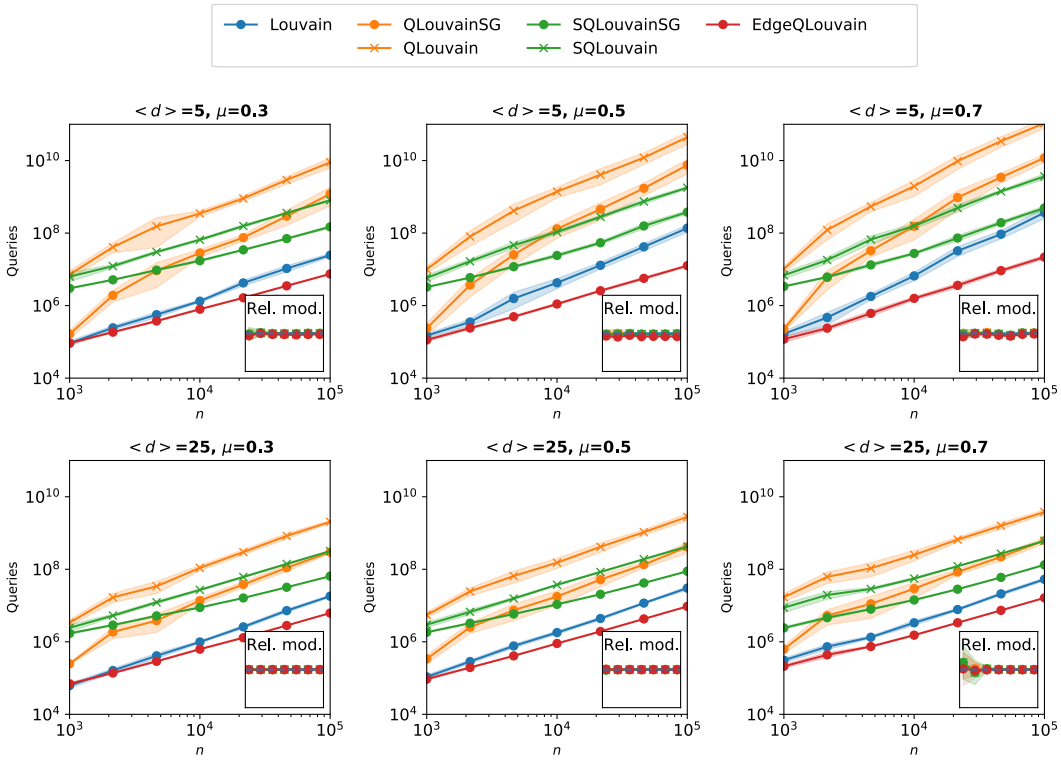

Fig. 2. Numerical results for the query counts of the classical and quantum Louvain algorithms from Section 3 on FCS-type graphs with a fixed community size $S = 50$. The average degree is either $\langle d \rangle = 5$ (top) or $\langle d \rangle = 25$ (bottom). The horizontal axis indicates the total number of nodes n and the vertical axis the number of queries made to the function g_Δ. Each data point corresponds to the average across 10 randomly generated graphs and the shaded area represents one standard deviation. In every sub-figure the bottom-right box plots the modularity relative to the one obtained with original Louvain (indicated with the dashed blue line) as function of n (logarithmic vertical axis). For the modularity, the limits of the y-axis are set at $\pm 10\%$ relative difference to Louvain.

5.1.1 Absolute query counts

Figs. 2 and 3 show the estimated average number of queries made and the modularities obtained by the classical and all of our quantum Louvain algorithms on FCS- and LFR-type networks with up to 10^5 nodes. In terms of absolute number of queries, we find that the sparse variants of **QLouvain** and **SimpleQLouvain** generally outperform their non-sparse counterparts and that all four are outperformed by the much simpler **EdgeQLouvain** algorithm. In fact, only **EdgeQLouvain** was able to achieve an observable speed-up over classical Louvain within the limits of our data sets – for $n > 2000$ it achieves this for all studied graph types and parameter combinations.

We also find that the quantum algorithms perform better relative to their classical counter-

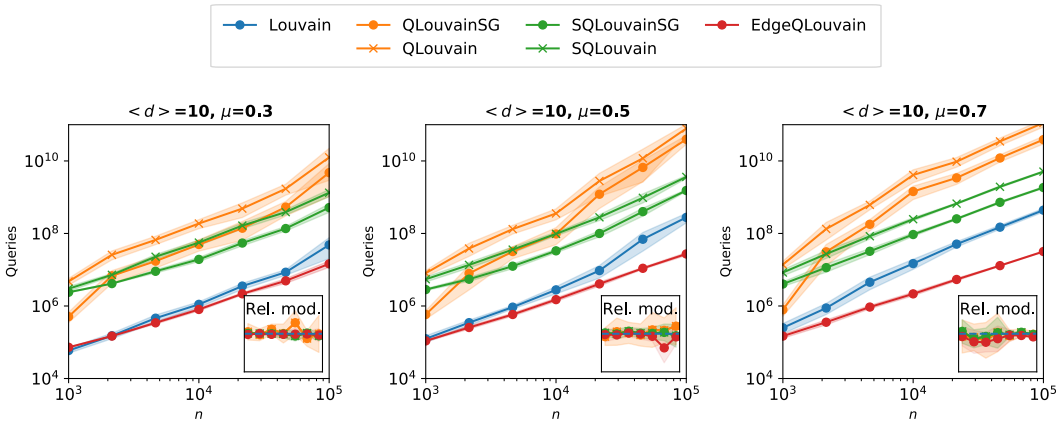

Fig. 3. Numerical results for the query counts of the proposed classical and quantum Louvain algorithms on LFR-type graphs with parameters $\tau_1 = 3$, $\tau_2 = 2$, $\langle d \rangle = 10$, $d_{\max} = 100$ and $S_{\max} = 100$. See the caption of Fig. 2 for a detailed description of the contents of individual plots and figures.

part when μ is large, which corresponds to graphs with relatively little community structure. An explanation for this is that for those graphs the fraction of good vertices could in general be smaller compared to graphs with high community structure, which corresponds to the regime where **QSearch** outperforms classical sampling. For the obtained modularities we found that for FCS-type graphs the relative difference in behaviour to the original Louvain algorithm is very small. The differences are more profound for the LFR-type graphs, in particular for **EdgeQLouvain**.

5.1.2 Estimating average case polynomial speed-ups

In Section 3 we showed analytically that, for a large collection of graph configurations, quantum algorithms for community-detection can achieve a polynomial speed-up over the classical Louvain algorithm on which they are based. In this section we use the data of Section 5.1.1 to estimate this polynomial speed-up, in expectation, for different graph configurations. Table 2 shows the coefficients corresponding to the linear fits of log-log plots of the data for the different algorithms as shown in Figs. 2 and 3. Recall that the ratio of the obtained coefficient as compared to Louvain is upper bounded by 2 – achieving this value would correspond to the full quadratic speed-up. For all studied graph configurations, **SimpleQLouvain** (both sparse and non-sparse) and **EdgeQLouvain** show (varying) polynomial speed-ups over Louvain. For FCS-type graphs, **SimpleLouvainSG** achieves the overall best scaling and for LFR-type graphs the best scaling is achieved by **EdgeQLouvain**. **QLouvain** almost always has a scaling that is at most comparable to Louvain, but this is mostly an artefact of the algorithm itself: for relatively small instances it predominantly uses classical routines; as n increases it begins to use more Grover steps – however, these Grover steps are still performed on relatively small lists. We expect that for much larger n one will also observe an asymptotic speed-up for **QLouvain** and **QLouvainSG**, though of course these sizes of n might not occur in practice.

		Degree weighted poly-fit (Estimated polynomial speed-up)					
	Configuration	OL	QLSG	QL	SQLSG	SQL	EQL
FCS	$\langle d \rangle = 5,\ \mu = 0.3$	1.23	1.76 (0.70)	1.45 (0.85)	0.86 (1.43)	1.07 (1.15)	0.96 (1.28)
	$\langle d \rangle = 5,\ \mu = 0.5$	1.5	2.07 (0.72)	1.69 (0.89)	1.07 (1.40)	1.23 (1.22)	1.03 (1.46)
	$\langle d \rangle = 5,\ \mu = 0.7$	1.71	2.17 (0.79)	1.89 (0.90)	1.12 (1.53)	1.37 (1.25)	1.15 (1.49)
	$\langle d \rangle = 25,\ \mu = 0.3$	1.24	1.44 (0.86)	1.34 (0.93)	0.80 (1.55)	1.06 (1.17)	0.98 (1.27)
	$\langle d \rangle = 25,\ \mu = 0.5$	1.21	1.42 (0.85)	1.29 (0.94)	0.85 (1.42)	1.08 (1.12)	1.01 (1.20)
	$\langle d \rangle = 25,\ \mu = 0.7$	1.13	1.37 (0.82)	1.14 (0.99)	0.87 (1.30)	0.92 (1.23)	0.95 (1.19)
LFR	$\langle d \rangle = 10,\ \mu = 0.3$	1.43	1.81 (0.79)	1.62 (0.88)	1.27 (1.13)	1.38 (1.04)	1.21 (1.18)
	$\langle d \rangle = 10,\ \mu = 0.5$	1.75	2.31 (0.76)	1.95 (0.90)	1.48 (1.18)	1.47 (1.19)	1.27 (1.38)
	$\langle d \rangle = 10,\ \mu = 0.7$	1.63	2.03 (0.80)	1.81 (0.90)	1.42 (1.15)	1.44 (1.13)	1.19 (1.37)

Speed-up factor
< 0.71
0.71 - 0.83
0.83 - 0.95
0.95 - 1.05
1.05 - 1.20
1.20 - 1.40
> 1.40

Table 2. Estimated polynomial degrees of the expected number of queries for the quantum algorithms and original Louvain on the FSC and LFR-type graphs. The central number in each cell corresponds the estimated polynomial degree obtained from a weighted fit of the form $an + b$, with the weights set at $\log n$, using the log-log data from Figures 2 and 3. The number in the top-right corner (in parentheses) estimates the polynomial speed-up, and is defined as the ratio of the estimated polynomial degree of Louvain and the respective quantum algorithm.

5.2 Real-world data-sets

Since analysis based on artificial networks only has limited value in predicting performance on actual real-world networks, we have also performed runs on large data-sets available at [23] and [28]. Table 3 shows the obtained modularities and total query count for a selection of our quantum algorithms. Both **QLouvain** and **QLouvainSG** are not considered here, as we found they were always outperformed by **SimpleQLouvain** and **SimpleQLouvainSG** on the artificial data sets. We find that for these instances **EdgeQLouvain** is able to achieve a modest speedup over **OL**, as well as obtaining slightly better modularities on average. Similar to the results obtained for artificial networks in Section 5.1, **SimpleQLouvainSG** is not able to achieve a speed-up on graphs with sizes of the orders of magnitude considered. Interestingly, in all but one case **SimpleQLouvain** outperformed **SimpleQLouvainSG** even though the real data sets have a very low average degree and hence are relatively sparse. This is due to the fact that the considered networks have in fact sometimes a very large maximum degree $d_{\max}$, which is the relevant graph parameter that determines the query complexity: both **SimpleQLouvain** and **SimpleQLouvainSG** scale with $\delta_{\max}$, however with the former having a quadratic improvement over the latter. The fact that the average degree is much lower than $d_{\max}$ in these data sets suggests that Variable time amplitude amplification might give a significant improvement (see Appendix C). Also, we found that all simulations of our quantum algorithms were not able to finish running within 5 days for the largest considered data set (IMDB). Therefore, further improvements need to be made to our simulation process to be able to study even larger problem instances.

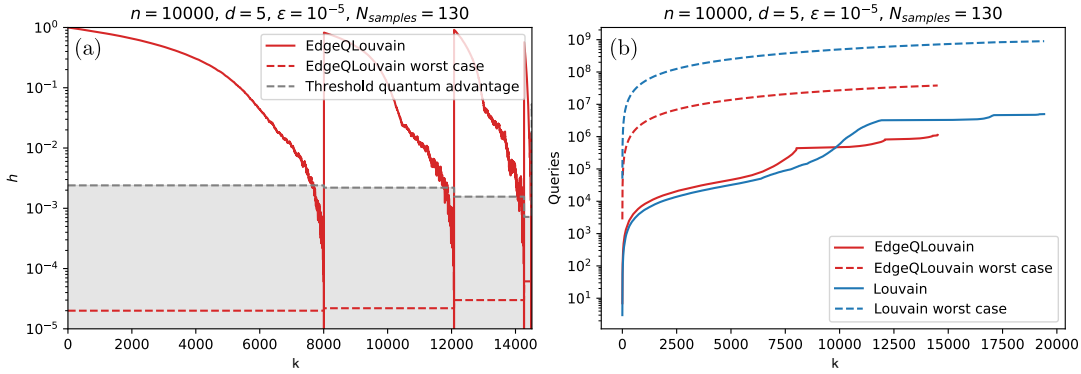

Fig. 4. (a) Fraction of marked items h vs step k of a run of the **EdgeQLouvain** algorithm. The gray dashed line represents the fraction (for a fixed list length, which changes after each phase of the algorithm) for which one obtains an absolute quantum speedup. The dashed red line represents the worst-case (i.e. when the quantum speedup is maximal) (b) Cumulative number of queries n vs step k for a run of **EdgeQLouvain** and Louvain. The dashed lines show the number of queries that would be made by each algorithm if every step represented the worst-case. The solid lines show the actual cumulative query count.

				Modularity			Total queries ($\times 10^7$)			
	Nodes	Edges	$d_{\max}$	OL	SQL(SG)	EQL	OL	SQL	SQLSG	EQL
Academia [23]	200k	1M	10693	0.6447	0.6456	0.6353	3.31	267	1180	1.08
DBLP [23]	317k	1M	343	0.8206	0.8210	0.8223	3.81	234	309	1.34
Amazon [23]	335k	925k	549	0.9262	0.9263	0.9264	2.09	195	118	1.21
Youtube [23]	496k	2M	25409	0.6825	0.678	–	3.35	743	6487	–
IMDB [28]	896K	4M	1590	0.6872	–	–	13.1	–	–	–

Table 3. Numerical results for the total amount of queries the classical and selected quantum Louvain algorithms make on real-world data-sets, averaged over five different runs. Entries with '–' timed out as they took longer than 5 days to compute.

5.2.1 Why is achieving a speedup difficult?

The quantum algorithms rarely (if at all) achieved a significant speedup over their classical counterparts. One explanation for this can be seen in Figure 4. In Figure 4(a), we see the fraction of marked items present in the search space (here, the number of directed edges $m = n\langle d\rangle$ in the graph) during a single run of the **EdgeQLouvain** algorithm. Only when the fraction is relatively small (dashed gray line) do we expect to achieve any advantage by using a quantum search in place of a classical one. As can be seen, there are relatively few steps during the run for which this is the case – for the remainder we in fact see a slowdown from using the quantum search subroutine. This behaviour is further demonstrated in Figure 4(b): we can see that for much of the run, the quantum algorithm is making more cumulative queries than the classical one, until it reaches a point at which a quantum speedup is achieved (i.e. when there are few marked items in the search space). This suggests that perhaps a better design for such algorithms could be to combine classical and quantum subroutines, and to switch to the quantum ones only when one expects to achieve some advantage.

5.3 Conclusion and discussion

In this paper we considered the framework of [9] for estimating the run-times of quantum algorithms that achieve modest polynomial speedups over their classical counterparts. As suggested there, in many cases, a traditional asymptotic analysis of the quantum algorithm is not informative enough to make decisions about whether, or for what input sizes, they might achieve a speedup over the best classical algorithm. In some cases, this is because a representative run-time cannot be obtained via such an analysis – something that is particularly true for (classical and quantum) heuristic algorithms. In others, it may be because the quantum algorithm is particularly sensitive to the input on which it is run, or just that the run-time obtained via an ordinary complexity analysis is not representative of the algorithm's true run-time.

To evaluate the usefulness of the approach we outlined in [9], we applied it here to a particular use-case of practical interest: community detection in large networks. Taking as a starting point a popular classical algorithm (the Louvain algorithm), we designed several quantum algorithms, each promising to give *some* speedup over the original. Using the bounds derived in [9], we obtained bounds for the quantum subroutines used by these algorithms, and then estimated the complexities of each algorithm for a number of randomly generated graphs, as well as some large real-world ones. We found that the algorithms whose analytically-derived asymptotic complexities were favourable were *not* the algorithms that obtained the lowest complexities in practice, nor the ones that scaled most favourably as a function of input size. This was perhaps not unsurprising, but does demonstrate that an analysis that goes beyond the usual asymptotic complexity one is necessary if one wishes to know whether a particular quantum algorithm could give a speedup for a task of practical interest, on an input representative of the ones it will receive in practice.

Our main observation when estimating the run-times of our quantum algorithms was that there is a large overhead associated with success probability amplification of quantum subroutines that is not present in the classical case, and that this can negate the speedup for even very large problem instances. This overhead is made more pronounced by the behaviour of Grover search when there are no marked items in the list: to verify that this is indeed the case, the quantum algorithm must perform a reasonably large number of repetitions of this Grover search sub-routine. Hence, in algorithms where we expect many of lists being searched over to in fact be empty (which was the case for one of our quantum algorithms, **QLouvain**), the quantum algorithm is often very slow in practice. Interestingly, this behaviour is not made apparent by the usual asymptotic run-time analysis of the algorithm.

Finally, we note that this kind of empirical run-time analysis is particularly useful for evaluation of quantum speedups of classical heuristics, such as those we consider in this work. In these cases, even if we know that the quantum algorithm achieves a per-step speedup over the classical algorithm, we will not know how much of this speedup survives when the algorithm is run to convergence, suggesting the need for an empirical approach to run-time estimation.

In the context of evaluating the potential of quantum algorithms in real-world settings, we argue that it would be useful to make the sorts of analyses that we perform in this work more commonplace. One way to do this would be to include the option of simulating certain quan-

tum algorithms, in the sense of this paper, within any of the existing quantum programming languages. Most of the work to do this would lie in proving good bounds on the run-times of various quantum primitives. Already for a couple of simple primitive we found this to be an extensive endeavour, but the upshot is that this work would only need to be performed once. With such tools at their disposal, we imagine that it would be easier for quantum algorithms designers to tailor their algorithms to particular tasks and datasets, and to more rapidly prototype ideas for quantum speedups without first undertaking an in-depth mathematical study.

Funding information CC was supported by QuantERA project QuantAlgo 680-91-034, with further funding provided by QuSoft and CWI. MF and JW were supported by the Dutch Ministry of Economic Affairs and Climate Policy (EZK), as part of the Quantum Delta NL programme. IN was supported by the DisQover project: a collaboration between QuSoft and ABN AMRO, and recieved funding from ABN AMRO and CWI.

Acknowledgements We would like to thank Harry Buhrman, Arjan Cornelissen, and Ian Marshall for helpful discussions, and Joran van Apeldoorn for providing tips on using Grover search to find the first item in a list. We Also thank Ton Poppe and Edo van Uitert from ABN AMRO for suggesting to study the use-case of community detection. The numerics of Section 5 were carried out on the Dutch national e-infrastructure with the support of the SURF Cooperative.

References

[1] Md. Ezaz Ahmed and Preeti Bansal, Clustering technique on search engine dataset using data mining tool, In *2013 Third International Conference on Advanced Computing and Communication Technologies (ACCT)*, pages 8689, 2013.

[2] Andris Ambainis, Quantum search with variable times, *Theory of Computing Systems*, 47(3):786807, 2010. arXiv:quant-ph/0609168.

[3] Andris Ambainis, Variable time amplitude amplification and quantum algorithms for linear algebra problems, 2012, arXiv:1010:4458.

[4] Simon Apers and Ronald de Wolf, Quantum speedup for graph sparsification, cut approx- imation and laplacian solving, In $\widehat{2}020$ IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS), pages 637648. IEEE, 2020. arXiv:1911.07306.

[5] Thomas Aynaud, python-louvain 0.15: Louvain algorithm for community detection, https://github.com/taynaud/python-louvain, 2020.

[6] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre, Fast unfolding of communities in large networks, *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, October 2008. arXiv: 0803.0476.

[7] Michel Boyer, Gilles Brassard, Peter Hoyer, and Alain Tapp, Tight bounds on quantum searching, *Fortschritte der Physik: Progress of Physics*, 46(4-5):493505, 1998. arXiv:quant-ph/9605034.

[8] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Gorke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner, On modularity clustering, *IEEE Transactions on Knowl- edge and Data Engineering*, 20(2):172188, 2008.

[9] Chris Cade, Marten Folkertsma, Ido Niesen, and Jordi Weggemans, Quantifying Grover speed-ups beyond asymptotic analysis, arXiv:2203.04975, 2022.

[10] Aaron Clauset, M. E. J. Newman, and Cristopher Moore, Finding community structure in very large networks, *Phys. Rev. E*, 70:066111, Dec 2004.

[11] Pasquale De Meo, Emilio Ferrara, Giacomo Fiumara, and Alessandro Provetti, Generalized louvain method for community detection in large networks, In *2011 11th international conference on intelligent systems design and applications*, pages 8893. IEEE, 2011. arXiv:1108.1502.

[12] Jordi Duch and Alex Arenas, Community detection in complex networks using extremal optimization, *Phys. Rev. E*, 72:027104, Aug 2005.

[13] Christoph Durr and Peter Hoyer, A quantum algorithm for finding the minimum, arXiv:quant-ph/9607014, 1996.

[14] Michelle Girvan and Mark Newman, Community structure in social and biological networks, In *Proceedings of the National Academy of Sciences of the United States of America*, 99:78216, 07 2002.

[15] Xiangquan Gui, Li Li, Jie Cao, and Lian Li, Dynamic communities in stock market, *Abstract and Applied Analysis*, 2014:19, 05 2014.

[16] Roger Guimera and Lus A. Nunes Amaral, Functional cartography of complex metabolic networks, *Nature*, 433(7028):895900, February 2005.

[17] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart, Exploring network structure, dynamics, and function using networkx, In *Gael Varoquaux, Travis Vaught, and Jarrod Millman, editors, Proceedings of the 7th Python in Science Conference*, pages 11-15, Pasadena, CA USA, 2008.

[18] Peter Hoyer, Arbitrary phases in quantum amplitude amplification, *Physical Review A*, 62(5):052304, 2000. arXiv:quant-ph/0006031.

[19] Pengsheng Ji and Jiashun Jin, Coauthorship and citation networks for statisticians, *The Annals of Applied Statistics*, 10(4):1779 1812, 2016.

[20] Andrea Lancichinetti and Santo Fortunato, Community detection algorithms: A comparative analysis, *Phys. Rev. E*, 80:056117, Nov 2009.

[21] Andrea Lancichinetti and Santo Fortunato, Community detection algorithms: a comparative analysis, *Physical review E*, 80(5):056117, 2009. arXiv:0908.1062.

[22] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi, Benchmark graphs for testing community detection algorithms, *Phys. Rev. E*, 78:046110, Oct 2008.

[23] Jure Leskovec and Andrej Krevl, SNAP Datasets: Stanford large network dataset collection, http://snap.stanford.edu/data, June 2014.

[24] M. E. J. Newman, Finding community structure in networks using the eigenvectors of matrices, *Phys. Rev. E*, 74:036104, Sep 2006.

[25] Mert Ozer, Nyunsu Kim, and Hasan Davulcu, Community detection in political twitter networks using nonnegative matrix factorization methods, In 2016 *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 8188, 2016.

[26] Xinyu Que, Fabio Checconi, Fabrizio Petrini, and John A Gunnels, Scalable community detection with the louvain algorithm. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 2837. IEEE, 2015.

[27] Jorg Reichardt and Stefan Bornholdt, Statistical mechanics of community detection, *Phys. Rev. E*, 74:016110, Jul 2006.

[28] Ryan A. Rossi and Nesreen K. Ahmed, The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[29] Sebastian A. Ros and Ivan F. VidelaCavieres, Generating groups of products using graph mining techniques, *Procedia Computer Science*, 35:730738, 2014. Knowledge-Based and Intelligent Information & Engineering Systems 18th Annual Conference, KES-2014 Gdynia, Poland, September 2014 Proceedings.

[30] V. A. Traag, L. Waltman, and N. J. van Eck, From Louvain to Leiden: guaranteeing well-connected communities, *Scientific Reports*, 9:5233, March 2019.

[31] Xiao-Ying Wang and Jonathan Garibaldi, A comparison of fuzzy and non-fuzzy clustering techniques in cancer diagnosis, In *Proc. of the Int. Conf. in Comput. Intell. in Med. and Healthcare*, 01 2005.

[32] Zhao Yang, Rene Algesheimer, and Claudio Tessone, A comparative analysis of community detection algorithms on artificial networks, *Scientific Reports*, 6, 08 2016.

[33] Christof Zalka, arXiv:quant-ph/9902049, 1999.

Appendix A The number of moves made by the Louvain algorithm

In this section we investigate the number of moves performed by the variants of the Louvain algorithm discussed in this paper as a function of the number of nodes n of the input graph. To start with, we provide a loose upper bound on the number of moves in Section A.1. Next, we numerically investigate the number of moves performed by the Louvain algorithm on actual datasets in Section A.2.

Appendix A.1. *A bound on the total number of moves*

First, we point out that there is a trivial upper bound on how long the Louvain algorithm takes to finish. Write T for the maximum possible number of moves that can be made before the (first phase of the) algorithm terminates. Since modularity is trivially bounded between two constants:

$$|Q| \leq \frac{1}{2W} \sum_{u,v \in V} A_{uv} + \frac{1}{4W^2} \sum_{u \in V} s_u \sum_{v \in V} s_v \leq 2,$$

and each move must strictly increase the modularity, it suffices to bound the smallest amount by which Q can increase after a single move.

Recall that the change of modularity when moving a vertex u from community $C_{\ell(u)}$ to community C_a is

$$\Delta_u^a = \frac{S_u^a - S_u^{\ell(u)}}{W} - \frac{s_u \left(\Sigma_a - \Sigma_{\ell(u)} + s_u\right)}{2W^2}.$$

The values in the numerators of the terms are sums over weights of edges, and W is the sum of all weights in the graph. Recall that we have $n = |V|$ vertices. If the weights on the edges are integers with $O(\log n)$-bit representations, then it is clear that the smallest non-zero value of Δ_u^a for $u \in V$ and $a \in [n]$ is $\Delta_{\min} = 1/W^2 = \frac{1}{|E|^2 \cdot O(poly(n))}$. For an unweighted graph, this is just $\frac{1}{|E|^2}$. Hence, the maximum number of moves T that can be made before there are no more moves that can increase modularity is $2/\Delta_{min} = |E|^2 \cdot O(poly(n)) = O(poly(n))$. For an unweighted graph, we have in particular $T = O(|E|^2)$.

Since $\frac{1}{p_k} \leq n$ (for every k), we can upper bound the time complexity of the Louvain algorithm by

- $\mathcal{O}(nd|E|^2)$ for unweighted graphs, and
- $\mathcal{O}(poly(n))$ for weighted graphs with weights expressed with $\log(n)$ bits.

We conclude that the Louvain algorithm is at worst a polynomial-time algorithm, although its run-time in practice will depend on the particular problem instance. In practice, the Louvain algorithm is mostly used for large n, small (constant) d sparse graphs.[o]

Appendix A.2. *The number of moves for actual datasets*

Next, we numerically investigate the number of moves performed by the variants of the Louvain algorithm discussed in this paper as a function of the number of nodes n of the input graph. Our results can be found in Fig. A.1.

Based on Fig. A.1 we find that for all quantum algorithms the amount of moves scale similarly in graph size n as compared to the Louvain algorithm, with the biggest differences occurring for graphs with a lower average degree $\langle d \rangle$. In agreement with [20], the data backs up the claim that the number of moves is $\mathcal{O}(n \log(n))$.

Appendix B Numerical results on the data structure

In Section 2.3.2 we mentioned that adding an extra data structure allows for a run time speedup of OL (original Louvain). In this section we show numerical results supporting this claim.

[o] See references in the Introduction.

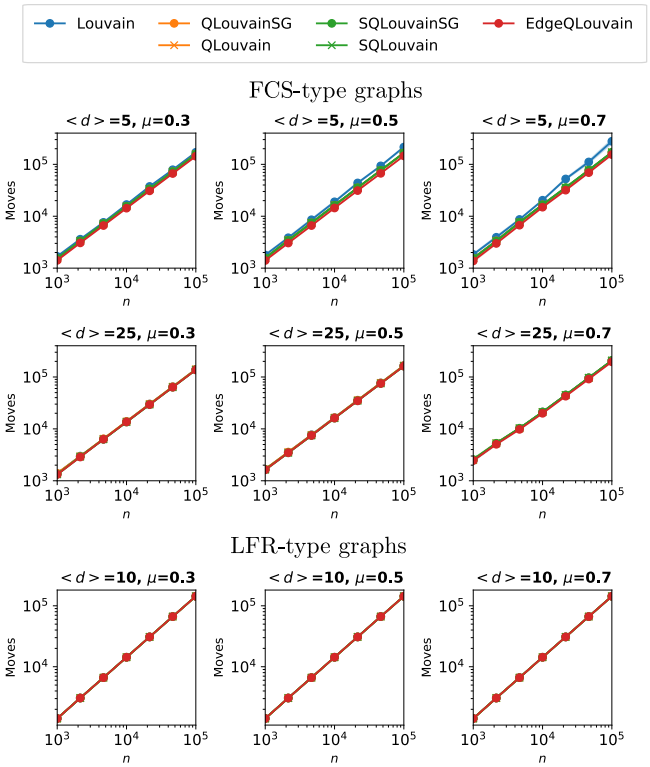

Fig. A.1. The average number of moves of all considered algorithms as a function of the graph size n.

We compare the run time between OL and OL with extra data structure using a python implementation of Louvain given by [5]. For OL we use exactly this implementation of louvain. For OL with extra data structure we changed the code to incorporate the data structure as described in Section 2.3.2. Run time tests are done in real time by tracking how long the algorithm takes to converge. The algorithms should in principle converge to a similar solutions since they differ only in their internal randomness. We also do a memory test, to see how much extra memory is used by introducing the extra data structure. This memory test registers the peak usage of memory.

The algorithms are tested on FCS- and LFR-type graphs as generated by the algorithm in Appendix D and the NetworkX implementation [17]. Every instance is run on 10 graphs, averaged and shown in Figure B.1. As predicted, OL with data structure slightly outperforms OL in run time when the average degree is low ($\langle d \rangle = 5$). This run time advantage is lost when the average degree increases ($\langle d \rangle = 25$). There is an extra constant overhead in memory when the extra data structure is introduced, as expected.

Appendix C Quantum Louvain with variable time amplitude amplification

In this appendix we describe in detail our quantum algorithm for community detection based on the technique of variable time amplitude amplification, and give a proof of its correctness and run-time. In [3], Ambainis describes a version of amplitude amplification for

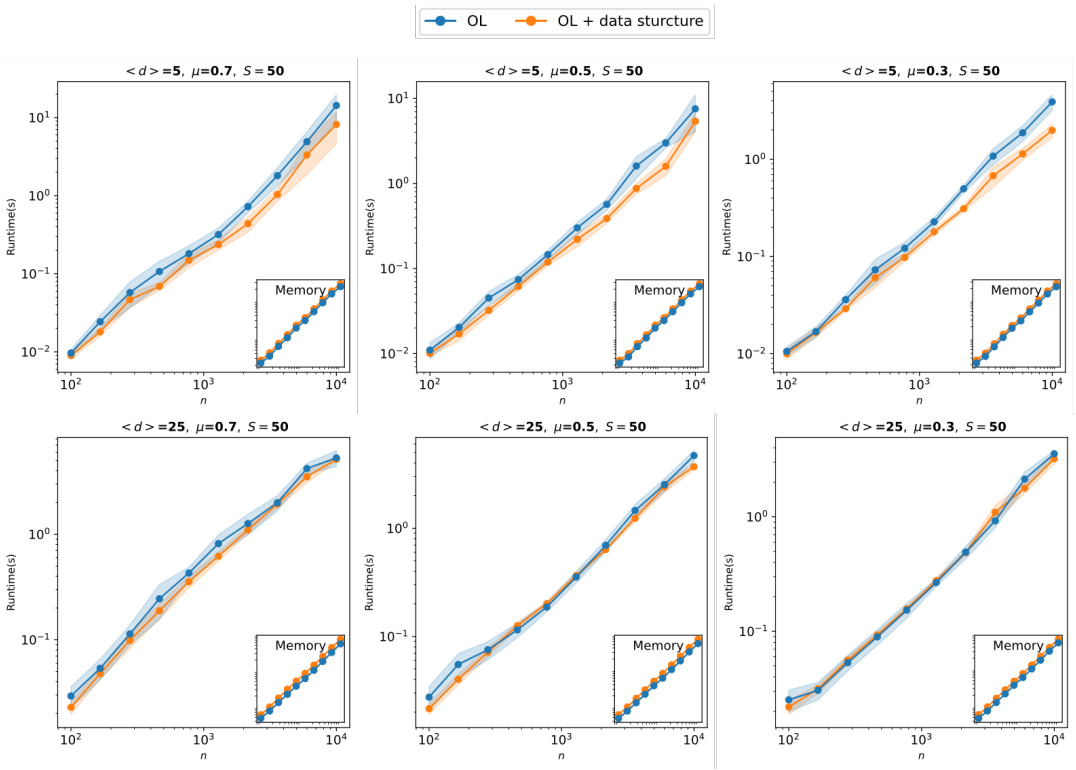

Fig. B.1. Numerical results for run time comparison between original Louvain and original Louvain with extra data structure. The algorithms were tested on FCS-type graphs with fixed community size $S = 50$, and average degrees $\langle d \rangle = 5$ (top) and $\langle d \rangle = 25$ (bottom). From left to right community structure is increased by decreasing μ. The horizontal axis indicates the amount of nodes n and the vertical axis the total run time (top). Each point corresponds to an average over 10 randomly generated graphs and the shaded area represents the standard deviation. The box in the bottom right shows peak memory usage (the number of bits).

the situation where the subroutine used by the quantum algorithm takes a different time to finish for each branch. This algorithm goes by the name variable time amplitude amplification (VTAA). Intuitively, VTAA executes amplitude amplification with a subroutine that can have different stopping times, with a final run-time that depends on some average of the individual stopping times, rather than being limited by the slowest branch as is the case in the algorithm presented in the previous section. We note that it is also possible to use variable time *Grover search*[p] [2] to obtain this behaviour, but here the algorithm assumes that there is only a single marked item and the run-time does not improve when there are multiple marked items, and hence the run-time of the algorithm will generally be quite poor, taking a time that is roughly $\tilde{O}(\sqrt{\delta_{\text{avg}}|L|})$ per move compared to the classical $O(\delta_{\text{avg}}/f)$, with k the fraction of good vertices present in L.

Here we use VTAA with a classical subroutine $\mathcal{A}$ that checks the neighbouring communities of a vertex one by one, requiring δ_u calls to the unitary for g_Δ and other operations for each vertex u. Let L be a list of vertices, and let $n \geq |L|$ be an upper bound on its size. The algorithm $\mathcal{A}$ acts on four registers, a $\log(n)$-sized vertex register, two $\log(\delta_{\max})$-sized neighbor-index registers, and 2-qubit flag register that can take the values 0, 1 and 2. The flag states correspond to: 0 *found no neighbor to move to*, 1 *found a neighbor to move to*, and 2 *still searching*. $\mathcal{A}$ consists of the sequence of unitaries: $\mathcal{A} = \underbrace{\mathcal{A}_c \cdots \mathcal{A}_c}_{\delta_{\max} \text{ times}} \mathcal{A}_s$, where

$$A_s |0\rangle |0\rangle |0\rangle |0\rangle = \frac{1}{\sqrt{|L|}} \sum_{u \in L} |u\rangle |1\rangle |0\rangle |2\rangle \tag{C.1}$$

sets up the initial state, and then each of the remaining $\mathcal{A}_c$'s sequentially check the neighboring communities of all vertices in superposition. The third register, which is initially set to 0, keeps track of the neighboring community index that currently maximises $\Delta_u^{\eta_u(j)}$. Here we use the convention that $\Delta_u^{\eta_u(0)} = 0$. Now, $\mathcal{A}_c$ acts on basis states as

$$\mathcal{A}_c |u\rangle |j\rangle |j_{\max}\rangle |f\rangle := \begin{cases} |u\rangle |j\rangle |j\rangle |2\rangle & \text{if} \quad f = 2, \quad j < \delta_u \quad \text{and} \quad \Delta_u^{\eta_u(j)} > \Delta_u^{\eta_u(j_{\max})} \\ |u\rangle |j+1\rangle |j_{\max}\rangle |2\rangle & \text{if} \quad f = 2, \quad j < \delta_u \quad \text{and} \quad \Delta_u^{\eta_u(j)} \leq \Delta_u^{\eta_u(j_{\max})} \\ |u\rangle |j\rangle |j\rangle |1\rangle & \text{if} \quad f = 2, \quad j = \delta_u \quad \text{and} \quad \Delta_u^{\eta_u(j)} > \Delta_u^{\eta_u(j_{\max})} \\ |u\rangle |j\rangle |j_{\max}\rangle |1\rangle & \text{if} \quad f = 2, \quad j = \delta_u, \quad \Delta_u^{\eta_u(j)} \leq \Delta_u^{\eta_u(j_{\max})} \\ & \text{and} \quad \Delta_u^{\eta_u(j_{\max})} > 0 \\ |u\rangle |j\rangle |j_{\max}\rangle |0\rangle & \text{if} \quad f = 2, \quad j = \delta_u, \quad \Delta_u^{\eta_u(j)} \leq \Delta_u^{\eta_u(j_{\max})} \\ & \text{and} \quad \Delta_u^{\eta_u(j_{\max})} = 0 \\ |u\rangle |j\rangle |j_{\max}\rangle |f\rangle & \text{if} \quad f = 0 \quad \text{or} \quad f = 1, \end{cases} \tag{C.2}$$

where $\eta_u(j) \in \zeta_u$ is the j-th neighboring community of u.

The algorithm $\mathcal{A}_c$ is just a coherent implementation of a classical algorithm that, given a vertex u, computes Δ_u^α for the neighboring communities $\alpha \in \zeta_u$ of u one by one. After visiting

[p]Here, we have a collection of n items $x_1, \ldots, x_n$ and we would like to find an $i : x_i = 1$. Let t_i be the number of time steps required to evaluate each x_i. Then variable time Grover search can find an i in time $\tilde{O}(\sqrt{t_1 + t_2 + \cdots + t_n})$. If there are multiple i's satisfying $x_i = 1$, then the algorithm actually becomes *slower* (by a constant factor).

all neighboring communities, $\mathcal{A}_c$ stops and puts in the fourth register either a 0, signifying that no neighbouring communities of u offer a good move, or otherwise puts a 1. Because different vertices will have a different number of neighboring communities, $\mathcal{A}_c$ has several different stopping times.

If we were to run $\mathcal{A}$ on the all zeros state, we would obtain the final state

$$\mathcal{A}\left|0\right\rangle\left|0\right\rangle\left|0\right\rangle\left|0\right\rangle = \frac{1}{\sqrt{|L|}}\sum_{u\in L}\left|u\right\rangle\left|\delta_u\right\rangle\left|j_u\right\rangle\left|\bar{\Delta}_u > 0?\right\rangle \tag{C.3}$$

$$= \alpha_0\left|\psi_0\right\rangle\left|0\right\rangle + \alpha_1\left|\psi_1\right\rangle\left|1\right\rangle =: \left|\psi_{\text{final}}\right\rangle\ , \tag{C.4}$$

where $|\psi_b\rangle$ is the state on the first three registers corresponding to the branch of the superposition in which the flag register is in state $|b\rangle$. The goal of amplitude amplification is to amplify the part of the state in which the flag register is in state $|1\rangle$, i.e. to amplify the amplitude $|\alpha_1|$ to $\geq 1/\sqrt{2}$, since this part of the superposition contains the indices of vertices for which a good move is available. The goal of *variable time* amplitude amplification is to achieve this amplification using a number of applications of the constituent unitaries $\mathcal{A}_c$ that takes into account the different stopping times for $\mathcal{A}$ on different branches.

Hence, our approach is to use the algorithm of Lemma 4, in which case the set of different stopping times $\{t_1, \ldots, t_m\}$ appearing in Eq. (5) for $\mathcal{A}_c$ will be the set of numbers of neighboring communities $\{\delta_u : u \in L\}$, and $T_{\max} = \delta_{\max}$ (this requires us to know $\delta_{\max}$ before applying the algorithm, but we note that this can easily be kept track of and updated after every vertex move). In order to be able to use VTAA, we need to check that $\mathcal{A}_c$ meets the necessary requirements outlined in [2]. To this end, for $i \in [\delta_{\max}]$ define $\mathcal{H}_i := \text{Span}\left(\{|u\rangle\,|j\rangle\,|*\rangle : u \in L, 1 \leq j \leq i\}\right)$, where $|*\rangle$ means there is no condition on the $j_{\max}$ register. Note that the subspaces $\mathcal{H}_i$ do not involve the flag register. Then we can prove Theorem 2, which is restated below for convenience.

Theorem 2 *Given a list L of vertices such that a fraction $f > 0$ of them are good, and the unitaries $\mathcal{A}_c$ and $\mathcal{A}_s$ defined in Eqs.* (C.1) *and* (C.2)*, we can use variable time amplitude amplification to construct a quantum algorithm* ***VertexFindVTAA****(L, ζ) that makes an expected*

$$O\left(\left(\delta_{\max}\log(\delta_{\max}) + \frac{t^q_{avg}}{\sqrt{f}}\log^{1.5}\delta_{\max}\right)\log(1/\zeta)\right)$$

calls to g_Δ, where

$$t^q_{\text{avg}} = \sqrt{\sum_{i=1}^{\delta_{\max}} p_i i^2},$$

and that returns the identity of a good vertex and the best move available to it with probability $\geq 1-\zeta$. If there is no good vertex, the algorithm will signal this and requires at most

$$O\left(\left(\delta_{\max}\log(\delta_{\max}) + t^q_{avg}\sqrt{|L|}\log^{1.5}\delta_{\max}\right)\log(1/\zeta)\right)$$

queries to do so.

proof. We will use the unitaries $\mathcal{A}_c$ and $\mathcal{A}_s$ defined above to construct the algorithm. In order to be able to apply Lemma 4, we first must check that $\mathcal{A}_c$ and $\mathcal{A}_s$ satisfy the various conditions described in [3]. In particular, we need to check that the following hold:

1. For $i \in [\delta_{\max} - 1]$, $\mathcal{H}_i \subseteq \mathcal{H}_{i+1}$.
2. For $i \in [\delta_{\max}]$, we should have that the state $|\psi_i\rangle$ obtained after i applications of $\mathcal{A}_c$ can be expressed as

$$|\psi_i\rangle = \underbrace{\mathcal{A}_c \cdots \mathcal{A}_c}_{i \text{ times}} \mathcal{A}_s |0\rangle |0\rangle |0\rangle = \alpha_{i,0} |\psi_{i,0}\rangle |0\rangle + \alpha_{i,1} |\psi_{i,1}\rangle |1\rangle + \alpha_{i,2} |\psi_{i,2}\rangle |2\rangle ,$$

   where $|\psi_{i,0}\rangle \in \mathcal{H}_i$, $|\psi_{i,1}\rangle \in \mathcal{H}_i$, and $|\psi_{i,2}\rangle \in (\mathcal{H}_i)^{\perp}$.
3. For $i \in [\delta_{\max}]$ and $P_{\mathcal{H}_i}$ the projector onto space $\mathcal{H}_i$, we have

$$P_{\mathcal{H}_i} |\psi_{i+1,0}\rangle = |\psi_{i,0}\rangle \quad \text{and} \quad P_{\mathcal{H}_i} |\psi_{i+1,1}\rangle = |\psi_{i,1}\rangle . \tag{C.5}$$

These conditions clearly hold for $\mathcal{A}_s$, and so we will focus on the unitary $\mathcal{A}_c$. Condition 1 holds by definition of the subspaces $\{\mathcal{H}_i\}_{i \in [\delta_{\max}]}$. In order to verify that condition 2 holds, given $i \in [\delta_{\max}]$, we observe the following.

- $|\psi_{i,0}\rangle$ is a superposition over vertices u for which $\delta_u \leq i$ and $\Delta_u^{\eta_u(j_{\max})} = 0$. In particular, for every u in the superposition, its neighboring community index is set to $\delta_u \leq i$, and hence $|\psi_{i,0}\rangle \in \mathcal{H}_i$.
- $|\psi_{i,1}\rangle$ is a superposition over vertices for which $\delta_u \leq i$ such that $\bar{\Delta}_u = \Delta_u^{\eta_u(j_{\max})} > 0$. In particular, for every vertex u in the superposition, its neighbor index is set to $\delta_u \leq i$ and hence $|\psi_{i,1}\rangle \in \mathcal{H}_i$.
- $|\psi_{i,2}\rangle$ is a superposition over vertices u for which $\delta_u > i$ (otherwise the flag would have been set to 0 or 1). In particular, all vertices u in the superposition have their neighbor index set to $i + 1$, and therefore$|\psi_{i,2}\rangle \in (\mathcal{H}_i)^{\perp}$.

For condition 3, we notice that when we apply $\mathcal{A}_c$ to $|\psi_i\rangle$, $\mathcal{A}_c$ only acts on $|\psi_{i,2}\rangle$, and then sets some flags for vertices u in the superposition $|\psi_{i,2}\rangle$ to 0 or 1. The vertices for which the flag was set to 0 or 1 all have their neighbor index set to $i + 1$. Thus, when we apply the projection opertor $P_{\mathcal{H}_i}$ to $|\psi_{i+1,0}\rangle$ or $|\psi_{i+1,1}\rangle$, these newly added vertices project to 0, and therefore Eq. (C.5) is satisfied.

Having verified that our subroutines $\mathcal{A}_c$ and $\mathcal{A}_s$ can be used inside VTAA, we turn our attention to the complexity of the resulting algorithm. Recall that f is the fraction of vertices in L that have a good move available (i.e. $\bar{\Delta} > 0$), and also let p_i be the probability that, for a randomly chosen vertex u, the number of neighbouring communities of u is i. Then the l_2 average over stopping times of $\mathcal{A}_c$ is

$$t_{\text{avg}}^q = \sqrt{\sum_{i=1}^{\delta_{\max}} p_i i^2}$$

and hence, using Lemma 4, we can apply VTAA directly to $\mathcal{A}_c$ and $\mathcal{A}_s$ to obtain an algorithm that produces the state

$$|\psi_{\delta_{\max}}\rangle = \alpha_{\delta_{\max},0} |\psi_{\delta_{\max},0}\rangle |0\rangle + \alpha_{\delta_{\max},1} |\psi_{\delta_{\max},1}\rangle |1\rangle$$

with $|\alpha_{\delta_{\max},1}|^2 \geq \frac{1}{2}$, by invoking $\mathcal{A}_c$ and its inverse at most

$$O\left(\delta_{\max}\log(\delta_{\max}) + \frac{t^{\mathrm{q}}_{\mathrm{avg}}}{\sqrt{f_k}}\log^{1.5}\delta_{\max}\right)$$

times. By measuring the second register, we will project the first register onto $|\psi_{\delta_{\max},0}\rangle$ with probability $\geq 1/2$, at which point we can measure it to obtain the identity of a good vertex, and the best move available to it, selected at random from the set of all good vertices. By repeating this process $O(\log(1/\epsilon))$ times, we will obtain a good vertex with probability $\geq 1-\epsilon$.

Finally, we note that every application of $\mathcal{A}_c$ requires $O(1)$ function calls to g_Δ (whilst $\mathcal{A}_s$ doesn't require any), and hence the number of function calls made by the algorithm is the same (up to constant multiplicative overhead) as the number of uses of $\mathcal{A}_c$ and its inverse. □

Using **VertexFindVTAA**, we can proceed to construct new VTAA-based versions of the algorithms for community detection described above. As an example, we can construct an analogue to **SimpleQLouvain**, whose run-time will now become

$$\sum_{k\in[T]} \tilde{O}\left(\delta_{\max} + \frac{t^{\mathrm{q}}_{\mathrm{avg}}}{\sqrt{f_k}}\right),$$

where the $t^{\mathrm{q}}_{\mathrm{avg}}$ is as in Theorem 2, and we choose $\zeta \leq 1/(3T)$ as the failure probability of **VertexFindVTAA**, to ensure that every step of the algorithm succeeds with high probability.

In contrast, a classical algorithm that searches for good vertices with replacement will make

$$t^{\mathrm{c}}_{\mathrm{avg}} = \sum_{i=1}^{\delta_{\max}} p_i i$$

function calls per move on average, leading to a classical run-time of

$$\sum_{k\in[T]} O\left(\frac{\delta_{\mathrm{avg}}}{f_k}\right). \tag{C.6}$$

The t^{q}_{avg} appearing in the quantum complexity is the '2-norm average' of the stopping times, rather than the 1-norm average of Eq. (C) that appears in the classical complexity. If the number of neighbouring communities is constant, then $t^{\mathrm{q}}_{\mathrm{avg}} = \delta_{\mathrm{avg}}$.

Appendix D Generation of FCS-type graphs

Algorithm 3 describes the algorithm we use to generate FCS-type random graphs.

Algorithm 3 FCS-type graph generation

function GRAPHGENERATIONFCS(n, S, μ, $\langle d \rangle$)
  Initialize a graph $G = (V, E)$ where $V = \{1, \ldots, n\}$ and $E = \emptyset$. Define community labels $L = \{1, \ldots, \lceil n/S \rceil\}$.
  Set $l_u = u \mod S$ for all $u \in \{1, \ldots, \lfloor n/S \rfloor\}$, and set $l_u = \lceil n/S \rceil$ for all $u \notin \{1, \ldots, \lfloor n/S \rfloor\}$. Let $V_l \subset V$ be the set of nodes in community l.
  Set $k = \langle d \rangle n$ as the counter of the remaining edges to be added.
  while $k > 0$ **do**
    pick $l \in L$ randomly uniform
    pick $u \in V_l$ randomly uniform, with probability $1 - \mu$ pick v from V_l uniformly at random , and with probability μ pick v from $V \setminus V_l$ uniformly at random.
    if $(u, v) \notin E$ **then**
      $E \leftarrow E \cup \{(u, v)\}$
      $k \leftarrow k - 1$
    end if
  end while
  return $\mathcal{G} = (V, E)$
end function