CS 860: Quantum Algorithms and Complexity **Spring 2006**

Lecture 01 — May 2, May 4

Lecturer: Andris Ambainis *Scribe: David L. Yeung*

0 Course Outline

0.1 Topics covered in this course

1. Quantum algorithms for graph problems
2. Quantum lower bounds
   - (a) versions of the adversary method
   - (b) limitations of the adversary method
   - (c) the polynomial method
   - (d) time-space lower bounds
3. Quantum random access codes
4. Quantum complexity classes
   - (a) quantum algorithms with advice
   - (b) quantum interactive proofs; quantum zero knowledge proofs
5. Quantum coin flipping

0.2 Course Requirements

The requirements for the students who are registered are:

1. Scribe notes for 2 to 3 lectures (20%),
2. Complete 2 to 3 homework assignments (40%),
3. Paper presentation: these will take place at the end of term, with one presentation per lecture period (40%).

1 Quantum Complexity of Graph Problems

1.1 Problem Definitions

Beginning in today's lecture, we will consider three problems on graphs, and examine the complexity of the quantum algorithms.

Given a graph G, we want to solve the following problems:

1. **Connectivity**
   We want to determine if G is connected (and find the connected components).

2. **Maximum/perfect matching**
   A matching of a graph is a set of edges of G with no vertices in common. A maximum matching is a matching that includes as many edges as possible. A perfect matching is one that covers all vertices of the graph. Note that a perfect matching is also a maximum matching. We want to determine if G has a perfect matching (or we may want to find the maximum matching).

3. **Triangle**
   A triangle in a graph is a set of three vertices with the property that there is an edge between each pair of them. We want to determine if a graph has triangles (and more generally, we want to determine if it has a clique of size k).

In this course, we will typically use the adjacency matrix representation for graphs. In particular, a graph G is specified by variables $g_{i,j}$ which are indexed by pairs of vertices (i, j):

$$g_{ij} = \begin{cases} 1 & \text{if } (i,j) \text{ is an edge,} \\ 0 & \text{if } (i,j) \text{ is not an edge.} \end{cases}$$

We access these variables by means of a "black box" that stores the graph and answers the query: $i, j \to g_{i,j}$. That is, we query the black box with a pair of vertices, and it tells us whether or not they are joined by an edge.

1.2 Complexity of the Problems

There are two ways of looking at the complexity:

1. **Query complexity**
   We count only the number of queries to the black box, and ignore how much time the overall computation takes.
2. **Time complexity**
   We are concerned with the overall computation time.

We want both the number of queries and the overall computation time to be small. We will primarily be concerned about the time complexity.

2 Computational Model

2.1 Classical algorithm with quantum subroutines

For our computational model, we will be using classical algorithms which will make calls to quantum subroutines. For the three algorithms we're considering, it turns out to be enough to define only a few quantum subroutines. We first describe these subroutines, and later we will describe the classical algorithms which makes calls to these subroutines. (We will not explain how these quantum subroutines are implemented. For such details, see, e.g., Nielsen and Chuang [NC00].)

2.2 Subroutine 1: Grover's search for *one* of k items

We have N boolean variables $X_1, \ldots, X_N \in \{0,1\}$, of which k have a value of 1 (we call these "marked"). In other words, $|\{i : X_i = 1\}| = k$ (where k is unknown).

The task is to find $i : X_i = 1$, that is, we want to find *one* of the k variables which is marked. There's a subroutine that, with probability $\geq 1 - \epsilon$,

- finds *some* $i : X_i = 1$ in $O\left(\sqrt{\frac{N}{k}} \log \frac{1}{\epsilon}\right)$ queries, if such an i exists, and
- outputs "`no such i`" after $O\left(\sqrt{N} \log \frac{1}{\epsilon}\right)$ queries, if no such i exists.

2.3 Subroutine 2: Grover's search for *all* k items

Instead of finding *one* of the k items, this variant of the above subroutine finds *all* k items:

- finds *all* $i : X_i = 1$ in $O\left(\sqrt{Nk} \log \frac{1}{\epsilon}\right)$ queries, if such i exist, and
- outputs "`no such i`" after $O\left(\sqrt{N} \log \frac{1}{\epsilon}\right)$ queries (the same as in Subroutine 1), if no such i exists.

Note that the complexity of the above subroutines is given in terms of queries. The time complexity is essentially the same (with possibly an extra factor of $\log N$).

3 Problem 1: Graph Connectivity

3.1 The Breadth-First Search (BFS) algorithm

The first of the three problems which we will study is graph connectivity. Given a graph G such as in Figure 1, we want to determine if G is connected (if there is a path from every vertex to every other vertex).

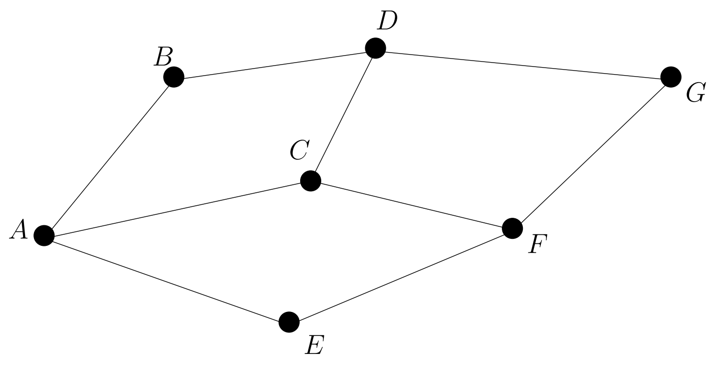

Figure 1. An example graph G which is connected.

The classical algorithm works by choosing an arbitrary vertex, say A, and building either a Breadth-First Search (BFS) tree or a Depth-First Search (DFS) tree starting at that vertex. If the tree does not include a particular vertex v, then we know that vertex v is not reachable from A.

The Breadth-First Search (BFS) algorithm:

```
1. Let $d(v) = -1$ for all $v \in V$.
   (Mark every vertex as "unvisited".)
2. Let $d(A) = 0$, $Q = \{A\}$.
   (Assign the starting vertex a distance of 0, and create a queue containing only it.)
3. While $Q$ is non-empty:
   (a) Let $u = \text{first}(Q)$.
       (Get the first vertex from the queue.)
   (b) Let $V = \{v \mid g_{u,v} = 1, d(v) = -1\}$.
       (Find all of its unvisited neighbours.)
   (c) For all $v \in V$, let $d(v) = d(u) + 1$.
       (Mark all the neighbours of $u$ as "visited" by assigning each of them a distance
       of 1 more than $u$.)
   (d) Add all $v \in V$ to the end of $Q$, and remove $u$.
```

Figure 2 shows the BFS built for G starting at vertex A. The algorithm evolves the queue Q as follows (the distance assigned to each vertex is given in superscript): $\{A^0\} \rightarrow \{B^1, C^1, E^1\} \rightarrow \{C^1, E^1, D^2\} \rightarrow \{E^1, D^2, F^2\} \rightarrow \{D^2, F^2\} \rightarrow \{F^2, G^3\} \rightarrow \{G^3\} \rightarrow \{\}$. The algorithm terminates when Q is empty.

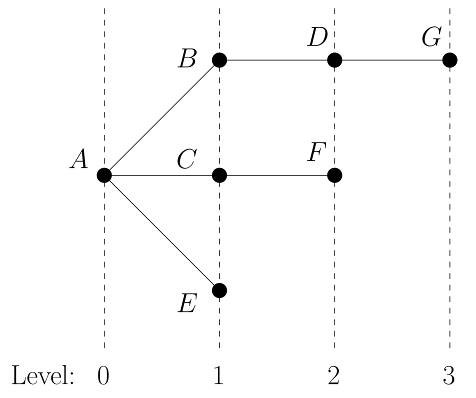

Figure 2. The Breadth-First Search (BFS) tree for the graph shown in Figure 1.

What is the complexity of this algorithm? The most expensive step is Step 3(b) – the complexity of this step is the same as the number of vertices n, and thus Step 3(b) takes $O(n)$ time. Furthermore, Step 3 is repeated n times. Therefore, the total running time of the classical algorithm is $O(n^2)$.

The idea behind the quantum version of this algorithm (due to Dürr et al. [DHHM04]) is to take Step 3(b), and substitute Grover's search in its place. We introduce the notation $\tilde{O}(\cdot)$ to mean that we ignore a polylog factor in the running time, i.e., we've dropped polynomials in $\log n$.

Theorem 3.1. *The BFS tree can be found by a quantum algorithm in time* $\tilde{O}(n^{\frac{3}{2}})$

Proof: Substitute Subroutine 2 in Step 3(b). Formally, we define the variables:

$$X_v = \begin{cases} 1 & \text{if } g_{u,v} = 1,\ \text{d(v) = -1} \\ 0 & \text{otherwise} \end{cases}$$

Then we run the quantum subroutine on these variables. Each variable can be checked in 2 operations (we need to check that $g_{u,v} = 1$ and $d(v) = -1$), so the total running time for this step is the same order as for Subroutine 2. □

3.2 Running Time Analysis

What's the running time of the quantum algorithm?

Let n be the number of vertices in G, and let V be the set of vertices found in Step 3(b) (then $|V|$ is the number of vertices found). Note that for the quantum version of Step 3(b), the order of the number of queries is:

- $O\left(\sqrt{n|V|}\log\frac{1}{\epsilon}\right)$ if $|V| \geq 1$, and
- $O\left(\sqrt{n}\log\frac{1}{\epsilon}\right)$ if $|V| = 0$.

We can be more specific. Let V_u be the set of unvisited neighbours V, found in Step 3(b), for the vertex u. Since each vertex $u \in V$ will be added to the queue Q once in Step 3, the sum $\sum_u \sqrt{n|V_u|}$ will give us a bound for the running time of the whole algorithm:

$$\begin{aligned}
\sum_u \sqrt{n|V_u|} &= \sqrt{n}\sum_u \sqrt{|V_u|} \text{ (since } n \text{ is independent of } u\text{)} \\
&\leq \sqrt{n}\sqrt{n}\sqrt{\sum_u |V_u|} \text{ (by the Cauchy-Schwartz inequality)} \\
&\leq \sqrt{n}\sqrt{n}\sqrt{n} \text{ (each vertex is visited at most once, so } \sum_u |V_n| \leq n\text{)} \\
&\leq n^{\frac{3}{2}}
\end{aligned}$$

Thus, we have that $\sum_u \sqrt{n|V_u|} \leq n^{\frac{3}{2}}$, and it is easy to see that the running time of all the times that Step 3(b) is invoked is $O(n^{\frac{3}{2}}\log\frac{1}{\epsilon})$. Every other step in the algorithm takes less time than this, so the running time of the algorithm is dominated by the running time of Step 3(b).

There is one more point to remember. We must choose ϵ correctly so that the algorithm works. We are invoking the quantum subroutine n times, and we want a high probability that the subroutine works every time. We can choose $\epsilon = \frac{1}{3n}$. Then the probability that the subroutine is incorrect in one call is less than $\frac{1}{3n}$, and the probability that it fails in (at least one of) n calls is at most $\frac{1}{3n}n = \frac{1}{3}$.

Thus, we can find the BFS tree for a graph with the above quantum algorithm, and since graph connectivity reduces to BFS (and checking every vertex), we can check for graph connectivity with a quantum algorithm in $\tilde{O}(n^{\frac{3}{2}})$ time.

4 Problem 2: Matching

4.1 Definition and Previous Results

Given a graph G, a **matching** of the graph is a set of edges with the property that every vertex is incident to at most one edge.

A **maximum matching** is a matching that has the largest possible number of edges, and a **perfect matching** is one in which every vertex is incident to exactly one edge. (Note that this means every perfect matching is also a maximum matching – but the converse is not true.) Figure 3(a) shows two matchings for a graph.

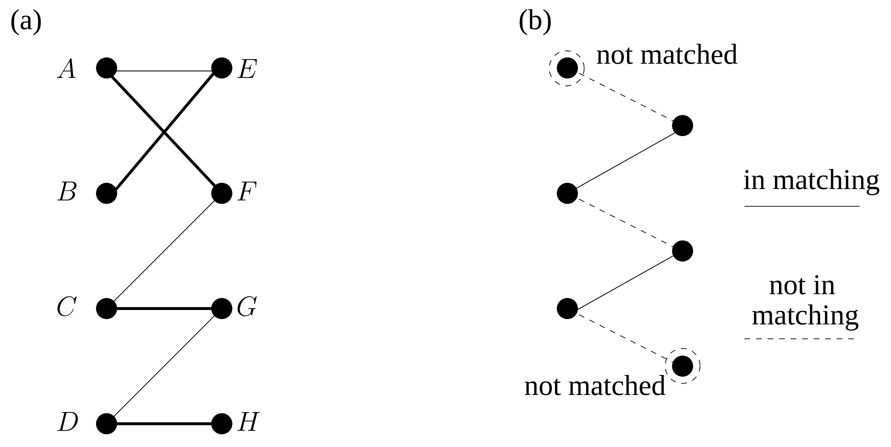

Figure 3. (a) A bipartite graph G with two matchings shown. The matching shown in bold is maximum. (b) An augmenting path – a path that begins and ends in a free vertex and alternates between free and matched edges.

The best classical algorithm for finding maximum matchings is due to Mucha and Sankowski [MS04] and has running time $O(n^{\omega})$, where ω is the exponent of the best matrix multiplication algorithm. Currently, the best running time is order $O(n^{2.376\cdots})$.

We will work with an older (and simpler) algorithm, due to Hopcroft and Karp [HK73], with running time $O(n^{2.5})$. We will derive a quantum algorithm from this algorithm, using the same ideas as before, with the resulting running time of $\tilde{O}(n^2)$.

4.2 The Hopcroft-Karp Algorithm

The basic idea is to search for an **augmenting path**, which is a path that begins and ends in a vertex not incident to any edge in the matching, and alternates between free and matched edges. Figure 3(b) shows an example of an augmenting path, which has 2 matched edges and 3 unmatched edges. Note that we can remove the 2 edges from the matching and add the 3 edges in their place, which increases the size of the matching by 1.

Lemma 4.1. *A matching is not maximum $\Leftrightarrow$ there exists an augmenting path.*

Proof: The $\Leftarrow$ direction is clear, since we can increase the size of the matching by 1 as in the above example.

We will prove the $\Rightarrow$ direction. Suppose we are given a graph G, such as in Figure 4, with a non-maximum matching M. Since M is not maximum, there exists some maximum matching M' larger than M.

Now, consider the connected components in $M \cup M'$. Note that every vertex is incident to at most 2 edges in $M \cup M'$ (since every vertex is incident to at most 1 edge in each of

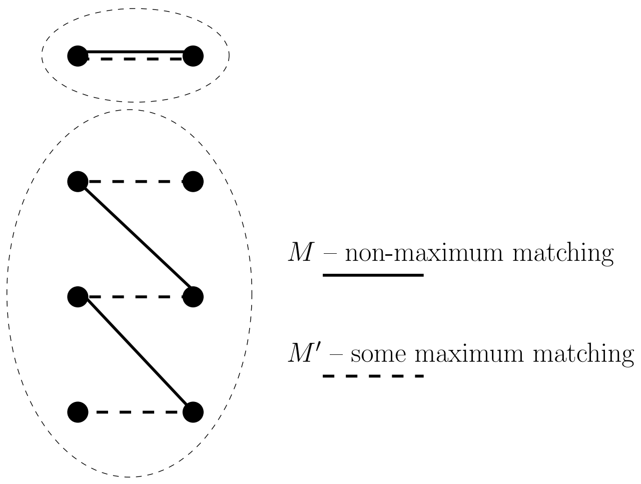

Figure 4. A graph with 2 connected components, a non-maximum matching M and a maximum matching M'.

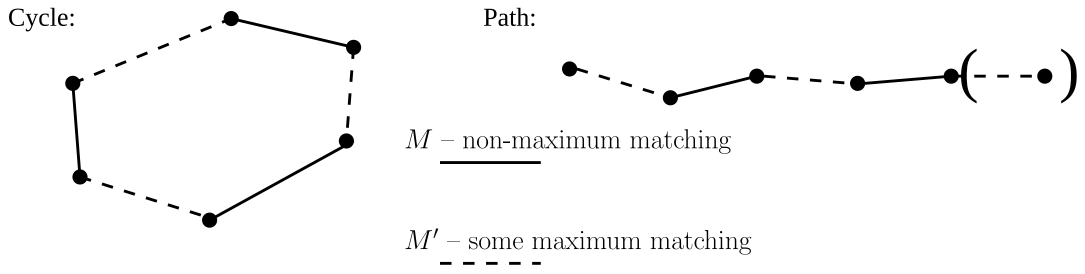

Figure 5. Each connected component in $M \cup M'$ is either a cycle or a path.

M and M'). Therefore, each connected component is either a cycle or a path. Figure 5 illustrates the two cases.

If a connected component is a cycle, it must have the same number of edges from M' as from M. On the other hand, if a connected component is a path, then it either has the same number of edges from M' as from M, or it has *exactly one more* edge from one than from the other. Since M' is a maximum matching while M is not, M' has more edges than M and there must be at least one augmenting path in some connected component of $M \cup M'$. $\square$

In fact, we can improve this lemma. If we have a lot of edges missing in our matching, the augmenting paths will be small.

Lemma 4.2. *If we have a non-maximum matching M with k fewer edges than the maximum matching M', that is, if $|M| \leq |M'| - k$, then there is an augmenting path of length at most $\frac{n}{k} - 1$.*

Proof: The proof is essentially the same as the above, but with more bookkeeping. Again, we look at the connected components of $M \cup M'$. In each of these connected components, the number of edges from M and M' will either be the same, or there will be one more edge from M'. Thus, there are at least k components with one more edge from M' than from M (these are augmenting paths).

The largest possible matching is a perfect matching, which matches every vertex, so $|M'| \leq \frac{n}{2}$. Since there are at least k augmenting paths, there exists an augmenting path with at most $\frac{|M'|}{k} \leq \frac{n}{2k}$ edges from M'. This augmenting path has length at most $\frac{n}{k} - 1$. □

We are now prepared for the (classical) Hopcroft-Karp matching algorithm. We use the expression "add the augmenting path to the matching" as a short-hand to mean that the edges in the augmenting path already in the matching should be removed, and the edges in the augmenting path not in the matching should be added.

The Hopcroft-Karp Matching Algorithm [HK73]
While there exists an augmenting path, repeat the following:

1. Let d be the minimum length of an augmenting path.
2. Find a maximal set of augmenting paths of length d.
   (That is, find a set of augmenting paths after adding which there are no more augmenting paths of length (at most) d.)
3. Add those augmenting paths to the matching.

Note that Step 1 is just a Breadth-First Search (BFS), while Step 2 is a Depth-First Search (DFS), as we will show below in Section 4.3. Classically, each of Steps 1 and 2 takes time $O(n^2)$, but in the quantum case they can be implemented in time $\tilde{O}(n^{\frac{3}{2}})$. So the running time of the algorithm is just that multiplied by the number of repetitions of the main cycle of the algorithm.

Lemma 4.3. *The number of repetitions of the main loop of the Hopcroft-Karp algorithm is $O(\sqrt{n})$ (in fact, it is at most $2\sqrt{n}$).*

Proof: By Lemma 4.2, if we have a non-maximum matching M with $\sqrt{n}$ fewer edges than the maximum matching M' (that is, $|M| \leq |M'| - \sqrt{n}$), then there is an augmenting path of length at most $\frac{n}{\sqrt{n}} - 1 = \sqrt{n} - 1 < \sqrt{n}$. By the contrapositive of this lemma, if the shortest augmenting path has length $\sqrt{n}$ or greater, then $|M| > |M'| - \sqrt{n}$.

In every iteration of the main loop of the Hopcroft-Karp algorithm, we are adding a maximum set of augmenting paths, so the length of the shortest augmenting path d is increasing with each iteration: $d = 1, 2, \ldots, \sqrt{n}$. In particular, after $\sqrt{n}$ iterations we have $d > \sqrt{n}$, at which point (by the contrapositive of Lemma 4.2) there are fewer than $\sqrt{n}$ edges missing in the current matching M. It then takes at most another $\sqrt{n}$ iterations to find the perfect matching. □

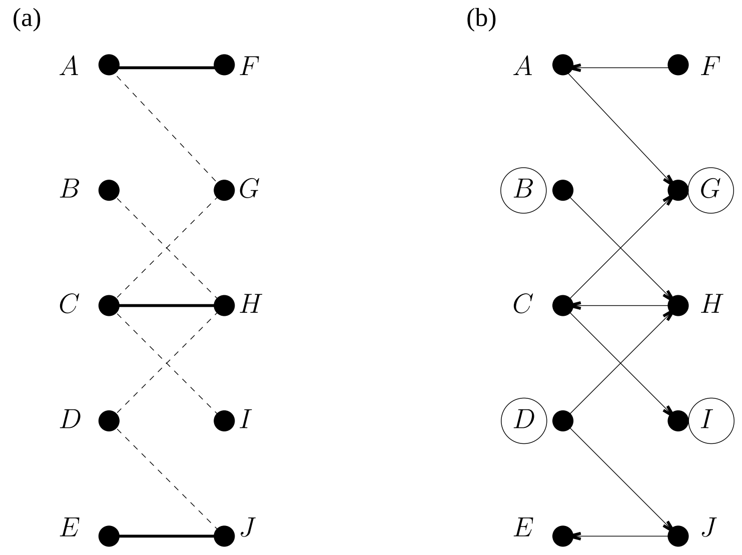

Figure 6. (a) A graph with a (non-maximum) matching. Solid edges are in the matching and dotted edges are free. (b) The graph with orientations assigned to the edges. An augmenting path must begin at B or D and end at G or I.

4.3 Finding augmenting paths

Figure 6(a) shows a graph with a (non-maximum) matching of size 3. To increase the size of this matching, we must do 2 things: (1) find the shortest augmenting path, and then (2) find a maximum set of augmenting paths of the same length. We will show how the first of these reduces to BFS, while the second reduces to DFS. We've shown above that BFS can be implemented in time $\tilde{O}(n^{\frac{3}{2}})$ in the quantum model, and we will show below that the same time complexity holds for DFS.

To begin, we create a directed graph by assigning edge directions: for example, we make edges in the matching point to the left and edges not in the matching point to the right. The result is illustrated in Figure 6(b). Note that now an augmenting path is just a path respecting the directions of the graph, starting in an unmatched vertex on the left (one of B or D), and ending in an unmatched vertex on the right (one of G or I).

To find the shortest augmenting path, we build a BFS forest starting at the vertices B and D, as shown in Figure 7. The shortest augmenting path found in our example is $\{(B,H),(H,C),(C,G)\}$, with a length of 3. This can be found in time $\tilde{O}(n^{\frac{3}{2}})$ by the quantum algorithm of Section 3.1.

We now verify that DFS can also be implemented in $\tilde{O}(n^{\frac{3}{2}})$ time with a quantum algorithm.

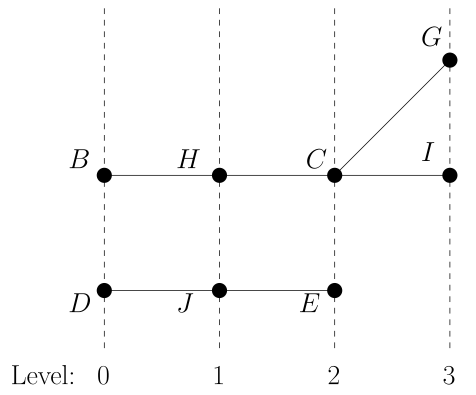

Figure 7. BFS forest for the graph shown in Figure 6.

4.4 The Depth-First Search (DFS) Algorithm

We show how quantum Subroutine 1 can be helpful in speeding up DFS. The algorithm below finds a maximal set of paths of length d. In particular, paths within a set are vertex-disjoint (each vertex belongs to at most 1 path).

The Depth-First Search (DFS) algorithm:

```
1. Let $u(v) = -1$ for all $v \in V$.
   (Mark every vertex as "unvisited".)
2. Repeat while there exists $s : \text{level}(s) = 0, u(s) = -1 \ldots$
   (Pick an unmatched vertex on the left that hasn't been visited.)
   (a) Set $s_0 = s, l = 0, u(s) = 0$
       (Mark vertex as visited and try to build an augmenting path with it as a starting point.)
   (b) Repeat until $l = -1$:
       i. If $l = d$, add $\{(s_0, s_1), (s_1, s_2), \ldots, (s_{d-1}, s_d)\}$ to the matching (and go back to start of Step 2).
          (If an augmenting path of length $d$ has been found, add it to the matching.)
       ii. Otherwise, use Subroutine 1 to find $v : \text{level}(v) = l + 1, g_{s_l,v} = 1, u(v) = -1$.
          (Find a neighbour on the next level which hasn't been visited yet.)
       iii. If found, set $s_{l+1} = v, u(v) = 0, l = l + 1$.
          (Increase the depth of the search.)
       iv. Otherwise, let $l = l - 1$.
          (Go back one level of depth and try another branch.)
```

The DFS algorithm terminates after all augmenting paths of length d have been found and added to the matching.

4.5 Running Time Analysis

The analysis of the DFS algorithm is somewhat similar to that of the BFS algorithm we analysed earlier in Section 3.2. We want to bound the complexity of Step 2(b)ii; the complexity of every one of the the other steps is $O(n)$.

For each particular vertex v, Step 2(b)ii may be performed multiple times.

Lemma 4.4. *The time for all runs of Subroutine 1 with $s_l = v$ is*

- $O\left(\sqrt{nd_v}\log\frac{1}{\epsilon}\right)$, *where d_v is the number of successful runs,*
- $O\left(\sqrt{n}\log\frac{1}{\epsilon}\right)$ *if $d_v = 0$, i.e., there were no successful runs.*

Proof: Suppose that there are d_v successful runs. What's the running time for the ith run?

During the ith run, there are at least $d_v - i + 1$ vertices still left to be found, since 1 vertex is found in the current run and $d_v - i$ will be found in future runs. Thus, for ith run, the running time is $O\left(\sqrt{\frac{n}{d_v-i+1}}\log\frac{1}{\epsilon}\right)$.

Now we sum these over all i from 1 to d_v:

$$\sum_{i=1}^{d_v}\sqrt{\frac{n}{d_v-i+1}} = \sum_{i=1}^{d_v}\sqrt{\frac{n}{i}} \le 2\sqrt{nd_v}$$

□

In a sense, one can think of the multiple runs of Subroutine 1 as one run of Subroutine 2, distributed over time (i.e., the marked items not all found at once, but one-by-one).

To get the running time of the DFS algorithm, we have to sum up these running times over all vertices of G. In other words, we want to bound $\sum_v \sqrt{nd_v}$. First, it is easy to see that $\sum_{v:d_v=0}\sqrt{n} \le n\sqrt{n}$. Then:

$$\begin{aligned}
\sum_v \sqrt{nd_v} &= \sqrt{n}\sum_v \sqrt{d_v} \text{ (since } n \text{ is independent of } v\text{)} \\
&\le \sqrt{n}\sqrt{n}\sqrt{\sum_v d_v} \text{ (by the Cauchy-Schwartz inequality)} \\
&\le \sqrt{n}\sqrt{n}\sqrt{n} \text{ (each vertex can be chosen at most once, so } \sum_v d_v \le n\text{)} \\
&\le n^{\frac{3}{2}}
\end{aligned}$$

Thus, the quantum version of DFS takes time $O(n^{\frac{3}{2}}\log\frac{1}{\epsilon})$, for error ϵ.

Now we are ready to analyse the running time of the quantum version of the Hopcroft-Karp matching algorithm. Recall (Lemma 4.3) that the main loop of the algorithm makes

$O(\sqrt{n})$ repetitions. In each iteration, one call is made to each of the quantum BFS and DFS algorithms, both of which we have shown have running time $O(n^{\frac{3}{2}} \log \frac{1}{\epsilon})$. Choosing $\epsilon = \frac{1}{n^C}$ for a sufficiently large constant C, we see that the running time of the quantum matching algorithm is $\tilde{O}(n^2)$, compared to the classical Hopcroft-Karp algorithm which has running time in $O(n^{2.5})$.

5 A few related Open Problems

We briefly mention a few open problems related to the above:

1. **Improving the matching algorithm**
   Can the running time of $\tilde{O}(n^2)$ be improved?

2. **Quantum complexity of network flows**
   The network flow problem is closely related to the problem of maximum bipartite matching.

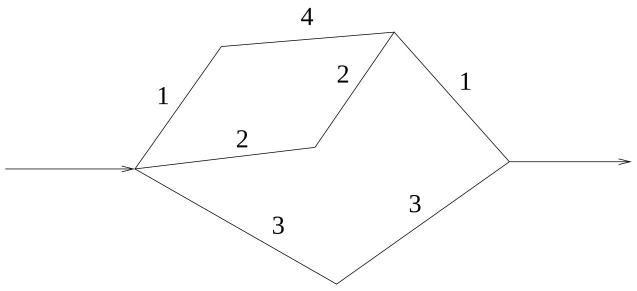

Figure 8. Graph of a single-source single-sink network flow.

   Suppose that we are given a flow network with n vertices, m edges, and integer capacities bounded by U, such as in Figure 8. The classical running time is dependent on m, n, and $\log U$. The fastest known quantum algorithm (due to Ambainis and Špalek [AŠ06]) has running time $O(\min(n^{\frac{7}{6}}\sqrt{m}U^{\frac{1}{3}}, \sqrt{nU}m) \log n)$.

   If $U = 1$ (that is, the capacities are either 0 or 1), then the quantum algorithm is better than any known classical algorithm. Can this be improved? It turns out that as the maximum capacity becomes larger than 1, the quantum algorithm becomes no better than the classical ones. Can one design a quantum algorithm so that the dependence on U is eliminated?

3. **Quantum algorithm for matrix multiplication**
   There is no graph structure in the matrix multiplication problem. However, the matrix multiplication problem is related to matching by the fact that the best classical

algorithm known for matching [MS04] uses matrix multiplication as a subroutine (see Section 4.1 above).

Thus, if matrix multiplication can be performed by a quantum algorithm in time less than $\tilde{O}(n^2)$, the running time of the quantum matching algorithm will also be improved.

4. **Quantum algorithm for deciding if** $\det(A) = 0$**?**
   We want to determine if $\det(A) = 0$ without necessarily computing it.

A little bit is known regarding possible approaches to tackling questions #3 and #4. Suppose that we're given $n \times n$ matrices A, B, and C, and we want to check if $AB = C$. The best quantum algorithm known (due to Buhrman and Špalek [BŠ06]) has running time $\tilde{O}(n^{\frac{5}{3}})$, while the best classical algorithm (a randomized algorithm given by Freivalds [Fre79]) has running time $O(n^2)$.

The idea behind the classical randomized algorithm is as follows: pick a random vector x, and check if $A(Bx) = Cx$ by right-multiplication. Clearly, if the equality $AB = C$ holds, then $ABx = Cx$ will also hold. On the other hand, if $AB \neq C$, then with high probability, $ABx \neq Cx$ for a random x.

In the next lecture, we will cover random walks and Markov chains, and develop a quantum version of random walk to solve the element k-distinctness problem. We will be building up to a quantum algorithm to solve the third of our three graph problems, namely, recognizing triangles (and more generally, k-cliques).

References

[AŠ06] Andris Ambainis and Robert Špalek. Quantum Algorithms for Matching and Network Flows. In *Proceedings of the 23rd International Symposium on Theoretical Aspects of Computer Science (STACS 2006)*, pages 172–183. Springer LNCS 3884, 2006.

[BŠ06] Harry Buhrman and Robert Špalek. Quantum Verification of Matrix Products. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2006)*, pages 880–889, Miami, Florida, USA, 2006. ACM Press.

[DHHM04] Christoph Dürr, Mark Heiligman, Peter Høyer, and Mehdi Mhalla. Quantum Query Complexity of Some Graph Problems. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004)*, pages 481–493, Turku, Finland, July 2004.

[Fre79] Rusins Freivalds. Fast Probabilistic Algorithms. In *Proceedings of the 8th Symposium on Mathematical Foundations of Computer Science (MFCS 1979)*, pages 57–69, 1979.

[HK73] John E. Hopcroft and Richard M. Karp. An $n^{\frac{5}{2}}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[MS04] Marcin Mucha and Piotr Sankowski. Maximum Matchings via Gaussian Elimination. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2004)*, pages 248–255, 2004.

[NC00] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, Cambridge, United Kingdom, 2000.