Lecture 13: Discrete time quantum walks

Rajat Mittal

IIT Kanpur

We have seen Grover's algorithm and amplitude amplification, giving a speedup on search based algorithms when the space is unstructured or the structure is not known. Classical random walks (Markov chains) have been used extensively to give improved algorithms in classical search problems where the search space is a graph. Is it possible to find speedups for such problems? Is there a natural analog of classical walks in the quantum world?

First thought might seem to give a negative answer, walks are designed to *get lost* in the search space, quantum processes are reversible. Though, it turns out that there indeed exists a concept of *quantum walks*, and our favourite algorithm (Grover search) can be framed in this language. Not just that, this framework can be used to give better algorithms in other cases as well. We will learn about classical random walks and the method to *quantize* them in this lecture note. The contents of this note is heavily based on the survey by Miklos Santha [3] and its exposition from Ronald de Wolf's course notes [1].

1 Random walk on a graph

A very general search problem is, there is a set X where we want to search and $M \subseteq X$ is the set of marked elements. We need to find a marked element in X. If there is no structure in X, only natural way is to pick an element from X uniformly at random and see if it is marked. To boost the error probability, we can repeat this procedure multiple times.

To be precise, if ϵ fraction of X is marked, we need to pick $O(1/\epsilon)$ elements from X to have constant probability of success. We saw in last few lectures, a quantum version of this algorithm only needs to be repeated $O(1/\sqrt{\epsilon})$ times (Grover search) to have constant probability of success.

Exercise 1. How much time will this strategy take if t out of n elements are marked?

Let us now understand how a random walk gives rise to a search algorithm. Instead of a plain set X, many a times we have some structure on X. In our case, say that there is a graph between the elements of X. In such cases, generally, moving from one vertex to another is considerably cheaper than picking an element uniformly at random (we will see such examples later in this lecture note). A natural search algorithm in such cases is:

1. Start in an arbitrary state/vertex (or a carefully chosen distribution on them).
2. Repeat *multiple times*: check if the element is marked, if not, *walk randomly for a few steps.*

This simple strategy has been helpful in classical algorithms (look at any literature on Markov chains for detail). Though, to talk about a formal proof of correctness and implementation, the algorithm above (random walk) needs to be defined more precisely. That means, how to take a step in a random walk (pick an arbitrary neighbour) and how many steps to take.

Generally, we can pick a neighbour uniformly at random. Formally, let us assume we are given a non-bipartite weighted connected graph $G = (X, E)$.

Note 1. These assumptions on the graph make the corresponding Markov chain ergodic and reversible. We won't worry about these definitions and Markov chains in general, though the results will be valid for all ergodic and reversible Markov chains.

The random walk on G is defined by the $|X| \times |X|$ transition matrix P, where

$$P_{x,y} = \frac{w_{x,y}}{\sum_y w_{x,y}}.$$

In other words, the row of P indexed by x defines a probability distribution according to the weights of edges around vertex x. The interpretation is, if we are at vertex x, the next step will take us to y with probability $P_{x,y}$.

Exercise 2. Why is a row a probability distribution?

How fast can such a random walk reach a marked vertex? What if we start in one *corner* of the graph and marked elements are in *some other corner*?

Note 2. Many of the results about classical random walk will just be stated (not proved) in this lecture note. You are welcome to look at standard references for a proof.

The properties of random walks have been extensively studied. It is known that irrespective of the starting state, the walk gets close to a steady state (we use the assumptions that graph is connected and not bipartite) after some *mixing time*. This mixing time and steady state only depend on eigenvalues and eigenvectors of the matrix P.

In particular, the highest eigenvalue (in terms of absolute value) of P is 1 and there is a unique unit eigenvector associated with it. Let us call that eigenvector π, it is going to be the steady state for random walk P. Let λ be the second highest eigenvalue in terms of its magnitude. We define $\delta = 1 - \lambda$ to be the spectral gap of P. The mixing time is of the order of $1/\delta$.

If the graph is regular then π is the uniform distribution over all vertices. This suggests the following algorithm.

1. Pick a state from the uniform distribution over X.
2. Repeat $O(1/\epsilon)$ times: check if the element is marked, if not, run the random walk for $O(1/\delta)$ steps.

Notice that the random walk is repeated $O(1/\delta)$ times so that it gets mixed again. We repeat the checking step $O(1/\epsilon)$ times, to boost the success probability to constant. How much time will this algorithmic framework take? How does it fare compared to the naive algorithm of picking a random element from X?

To analyze this, we need to define three costs: setup, update and checking.

- Setup (S): To sample an element from the uniform distribution on X.
- Update (U): To simulate taking one step of the random walk according to P.
- Checking (C): To check if an element/state is marked.

The naive classical algorithm to pick an element uniformly at random will have $\frac{1}{\epsilon}(S + C)$ cost. On the other hand, the random walk algorithm will have cost

$$S + \frac{1}{\epsilon}(C + \frac{1}{\delta}U).$$

You will prove this formally in the assignment.

Exercise 3. Convince yourself that depending upon various costs and parameters, either algorithm can outperform the other.

Is it possible to improve the dependence on ϵ, δ using a quantum computer? Remember our goal is to find a marked vertex.

2 Quantum walk on a graph

We want a similar framework for quantum random walk. To start with, matrix P is not unitary. The main questions to answer are:

- What is the equivalent unitary operator, call it $W(P)$, for quantum walk?
- What does it mean to *simulate* a quantum walk?

– Are there any advantages of doing a quantum walk instead of a random walk?

There is already *evidence* that the answer to the last question is yes. Grover search already shows a speedup when we ignore the graph structure. First, we will find a unitary walk operator $W(P)$. Using the walk operator $W(P)$, we will implement a quantum walk (MNRS framework [2]) and analyze its cost.

Note 3. There are many ways to define a quantum walk. The first quantum walk was proposed by Szegedy [4]. We have picked MNRS to illustrate the main concepts.

2.1 Quantum walk operator $W(P)$

As mentioned in the beginning, the idea of random walk is that it should *get mixed* in the graph. We saw later that indeed it mixes and gets to a steady state, irrespective of the starting state. In other words, it is not reversible.

Exercise 4. Construct a walk matrix P which is not unitary.

What then should be the quantum analogue of a random walk? One way out is to view our random walk in a different manner. Let us have a coin space, say set C, attached to the vertex space. Our state of the random walk will be the tuple (x, c), where $x \in X$ and $c \in C$. One step of the walk will consist of two operations: first one will be a *coin flip* and will change the second part of the state, and then a *shift operation* which will change the first part depending upon the second.

$$(x, c) \to (x, c') \to (y_{x,c'}, c').$$

Here, $y_{x,c'}$ denotes that y is the c'-th neighbour of x.

For a d-regular graph,

– Coin flip F: Pick a coin state uniformly at random out of $\{1, 2, \cdots, d\}$.
– Shift S: Label the neighbours of x with numbers from $\{1, 2, \cdots, d\}$ and take the i-th neighbour if the coin state is i.

Initial quantum walks were defined with $\mathbb{C}^{|C|\times|X|}$ as the state space. Now, we see that there is a more *useful* version. Instead of introducing a coin space, it makes sense to just use X as the second space. The state of the walk will be the tuple $(x, y) \in X \times X$. In that case, the shift operation S is basically an exchange, (x, y) goes to (y, x).

Exercise 5. What is the dimension of S matrix?

The coin flip should move according to the transition matrix P. Observe that in one step of the walk, the state x, u will go to y, x with probability $P_{x,y}$. That means, for the flip operation F, we should go from x, u to x, y with probability $P_{x,y}$. So F is a diffusion operation with control on the left state.

Exercise 6. Can you construct the flip matrix F now?

One step of the random walk is SF, two steps will be $SFSF$. You can notice that SFS is basically a diffusion with control on the right state. Summarizing, two steps of a random walk are same as $F'F$ where F' is the diffusion according to P with right state as control.

We can *quantize* these diffusion operations F, F'. Define $|p_x\rangle = \sum_{y\in X} \sqrt{p_{x,y}}|y\rangle$, first diffusion will be a reflection on the span of $|x\rangle|p_x\rangle$, second diffusion will be a reflection on the span of $|p_x\rangle|x\rangle$.

We can formally define the walk operator $W(P)$ to be $Ref(A)Ref(B)$. Here $A = span(|x\rangle|p_x\rangle : x \in X)$ and $B = span(|p_x\rangle|x\rangle : x \in X)$. How do we implement the walk operator, for $Ref(A)$, notice that $|x, 0\rangle \to |x, p_x\rangle$ can be thought of as a step of a classical random walk. We can first move $|x, p_x\rangle$ to $|x, 0\rangle$, perform reflection around $|x, 0\rangle$ and then reverse the first operation.

Exercise 7. How will we perform $Ref(B)$?

In other words, $W(P)$ takes constant number of walk operations. The walk operator is based on Szegedy's work on quantizing Markov chains [4]. He was also able to show the relationship between the spectrum of $W(P)$ and the spectrum of P. For us, the most important result is, $|\pi\rangle := \sum_{x \in X} \sqrt{\pi_x}|x, p_x\rangle$ is the unique 1 eigenvector for $W(P)$ and its spectral gap is more than $2\sqrt{\delta}$. Remember that π was the 1 eigenvector for P and δ was its spectral gap. We already see hint of a quadratic speedup here. We will assume these results without proof.

From the walk operator, we need to get the implementation of a quantum walk. We will see MNRS framework to do this, it uses phase estimation of this walk operator $W(P)$.

2.2 MNRS walk framework

We want a quantum equivalent of the random walk algorithm described in the first section (take steps to make walk mixed, sample, repeat). From the walk operator, the walk has to be performed in the space $\mathbb{C}^{|X| \times |X|}$. This means that our basis states will be $|x, y\rangle$ where $x, y \in X$. Taking cue from Grover search, define *good* basis states to be $|x, y\rangle$ where x is marked.

Let $|\pi\rangle := \sum_{x \in X} \sqrt{\pi_x}|x, p_x\rangle$ be the state defined by the unique eigenvalue 1 eigenvector of $W(P)$. Our target will be to move $|\pi\rangle$ to its projection on the good states,

$$|\mu\rangle = \frac{1}{c} \sum_{marked\ x} \sqrt{\pi_x}|x, p_x\rangle.$$

Exercise 8. What is c here?

Grover's algorithm tell us that this can be done with repeated applications of $Ref(\pi)Ref(\mu^\perp)$. The MNRS walk algorithm is,

1. Make the starting state $|\pi\rangle$.
2. Perform $O(1/\sqrt{\epsilon})$ times the operation $G = Ref(\pi)Ref(\mu^\perp)$.

Even though this algorithm looks very similar to the random walk algorithm, there is no mention of a quantum walk or a walk operator.

Exercise 9. What details are not present in the description of the algorithm above?

Even though states and vectors have been defined, we don't know how to perform the rotation $G = Ref(\pi)Ref(\mu^\perp)$. Recall that $\mu^\perp$ is basically the set of unmarked states on the first register, so an oracle call will do the required reflection. In other words, 1 checking operation is used to perform the reflection about $\mu^\perp$.

We will use the walk operator to perform reflection around π. A reflection around π should put a phase of -1 whenever the state is orthogonal to π and do nothing when the state is π. If we have a way of checking that the state is π, a CZ operation on the result of the checking mechanism will perform the reflection.

Here comes the phase estimation, π is the unique eigenvector with eigenphase 0. Suppose spectral gap ($\geq 2\sqrt{\delta}$) of $W(P)$ is big; we know that the eigenphase can be estimated. Whenever the estimated phase is considerably bigger than 0, we put a phase of -1. To be precise, whenever the phase is bigger than $\sqrt{\delta}$, we can put a phase of -1. This means that we only need to get the phase estimation with accuracy $\sqrt{\delta}$.

In steps, we first perform the phase estimation

$$|w, 0, 0\rangle \to |w, \theta, 0\rangle$$

Let $c_{\theta,\delta}$ be the indicator variable for $\theta \geq \delta$. Then we can change the last qubit by checking θ (even classically).

$$|w, \theta, c_{\theta,\delta}\rangle$$

Now, putting a CZ with control on the 3rd qubit (and reversing the previous operations)

$$(-1)^{c_{\theta,\delta}}|w, \theta, c_{\theta,\delta}\rangle \to (-1)^{c_{\theta,\delta}}|w, \theta, 0\rangle \to (-1)^{c_{\theta,\delta}}|w, 0, 0\rangle$$

As required, this will not do anything to π and put a phase of -1 to anything orthogonal to π.

Cost analysis: We will once again check the cost of this algorithm in terms of setup, update and checking cost.

- Setup (S): Cost to create state π.
- Update (U): Cost to perform $W(P)$.
- Checking (C): Cost to apply the checking oracle for marked elements.

The cost is $S+\frac{1}{\sqrt{\epsilon}}T$, where T is the cost of performing G. The first reflection requires 1 checking operation, the second reflection requires $O(\frac{1}{\sqrt{\delta}})$ $W(P)$ operations.

Exercise 10. By looking at the accuracy required for θ, show that we need to make $O(\frac{1}{\sqrt{\delta}})$ many $W(P)$ operations for the second reflection.

So, the total complexity is $O(S + \frac{1}{\sqrt{\epsilon}}(C + \frac{1}{\sqrt{\delta}}U))$. A quadratic improvement in both ϵ, δ as compared to the classical case.

2.3 Applications of MNRS walk framework

We have seen a general framework for quantum walk (MNRS) and its cost in terms of setup, update and checking. Let us look at search problems which can be framed as a random walk over a particular graph. By analyzing δ,ϵ and S, U, C costs, we will be able to give efficient quantum algorithms for these search problems.

Grover search: There is no structure, and the search can be modeled as a walk on a complete graph. The corresponding matrix $P = \frac{1}{n-1}(J - I)$, where J is all 1's matrix. You will show in the assignment that the spectral gap for P is $1 - \frac{1}{n-1}$ (approximately 1). That means the cost is $O(S + \frac{1}{\sqrt{\epsilon}}(C + U))$.

Exercise 11. What should be the query complexity of setup, update and checking?

The setup cost is of making the eigenvector π (uniform superposition in this case). So no queries are required to setup. Similarly, the walk only updates the indices (and does not need to track the value at the index), again zero cost.

The checking will take 1 query. In other words, the query complexity is $O(\frac{1}{\sqrt{\epsilon}})$, where ϵ is the fraction of marked elements. We recovered Grover search via MNRS framework.

Element distinctness problem: You are given a string $x \in [R]^n$, we view it as $x_1 x_2 \cdots x_n$. The element distinctness problem is to find if there exist i, j such that $x_i = x_j$. Notice that x_i's are not Boolean, we will assume R to be polynomial in n. We have query access to x (get x_i on asking index i). How many queries do we need to solve element distinctness problem?

What graph should we consider for this problem. The idea would be to keep a small set of indices (say r indices) and their corresponding values at a time. At each step, we will check if there is a collision inside our chosen set. If not, we will remove one index from the set and include another element. This process needs to be repeated enough times such that we find a collision with high probability (if there is one).

As suggested above, if we let a vertex contain the subset of r indices *and their value on* x. In other words, if S is $\{1, 2\}$, the corresponding vertex will be $\{1, 2, x_1, x_2\}$. The above algorithm corresponds to a random walk on $J(n, r)$ graph, called *Johnson graphs*. The vertices of Johnson graph are r size subsets of an n sized set. Two vertices are connected if they can be obtained from each other by exchanging one element. You can observe that the algorithm above was a random walk on the Johnson graph $J(n, r)$.

Exercise 12. Show that $J(n, r)$ is a regular graph (equal degree for all vertices) with degree $r(n - r)$.

Let us apply MNRS framework on $J(n, r)$ and analyze the cost.

If $r << n$, the spectral gap of $J(n, r)$ is around $1/r$. A vertex will be marked if it contains both the indices from the colliding pair. So, the fraction of marked elements is $O(r^2/n^2)$.

What are the setup, update and checking cost. Since $J(n, r)$ is a regular graph, its principal eigenvector is the normalized all 1's vector. So the starting state is an equal superposition of r sized pair of sets R, R' where R and R' are adjacent in $J(n, r)$. So the setup cost is $r + 1$. Since we already have the values of the indices on x, C is 0. The update requires us to calculate the corresponding values of R' from R (in superposition). That will take $O(1)$ queries.

So, our algorithm will take $O(r + \frac{n}{r}(0 + \sqrt{r} \cdot 1))$ queries.

Exercise 13. Show that the best r to choose is $n^{2/3}$.

Choosing $r = n^{2/3}$, we get an algorithm with cost $n^{2/3}$.

3 Assignment

Exercise 14. Prove that all eigenvalues of a transition matrix P will lie between 1 and -1.

Exercise 15. Read about Markov chains, steady state and mixing time.

Exercise 16. Convince yourself that the naive classical algorithm to pick an element uniformly at random will have $\frac{1}{\epsilon}(S + C)$ cost. On the other hand, the random walk algorithm will have cost

$$S + \frac{1}{\epsilon}(\frac{1}{\delta}U + C).$$

Exercise 17. Show that the eigenvalues of $\frac{1}{n-1}(J - I)$ are 1 and $\frac{-1}{n-1}$.

Exercise 18. Read more about Johnson graphs.

References

1. R. de Wolf. Quantum computig: lecture notes, 2019. `https://arxiv.org/abs/1907.09415`.
2. Frédéric Magniez, Ashwin Nayak, Jérémie Roland, and Miklos Santha. Search via quantum walk. *SIAM Journal on Computing*, 40, 08 2006.
3. M. Santha. Quantum walk based search algorithms. *TAMC*, 2008.
4. Mario Szegedy. Quantum speed-up of markov chain based algorithms. *Proceedings - Annual IEEE Symposium on Foundations of Computer Science, FOCS*, pages 32– 41, 11 2004.