Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.438 Algorithms For Inference
Fall 2014

Recitation 4: Elimination algorithm, reconstituted graph, triangulation

Contents

1 Types of queries on graphical models

Having studied definitions and properties of different types of graphical models, we may begin to about their applications. Toward this end, we study how to perform varous probabilistic queries on these models. In the context of solving these queries, we shall assume the graph structure and the set of potentials to be given.

There are 2 main types of queries- inference and MAP. (MAP stands for maximum a posteriori probability)

MAP query: Find $\arg\max_{x_1,x_2,\ldots x_n} \mathbb{P}_{\mathsf{X}_1,\mathsf{X}_2,\ldots \mathsf{X}_n}(x_1, x_2, \ldots x_n)$

Inference (marginalization) query: Find $\mathbb{P}_{\mathsf{X}_A|\mathsf{X}_B}(x_A|x_B)$, where $\mathsf{X}_A \cup \mathsf{X}_B \subseteq \mathsf{X}_V$.

Right now, we will concern ourselves with a special type of inference query, where A is a singleton set, and B is the empty set. That is, we would like to do the following:

Single node marginal probability query: Find $\mathbb{P}_{\mathsf{X}_\mathbf{i}}(x_\mathbf{i}) \forall x_i \in \mathcal{X}$, where i is an arbitrary node in the graph.

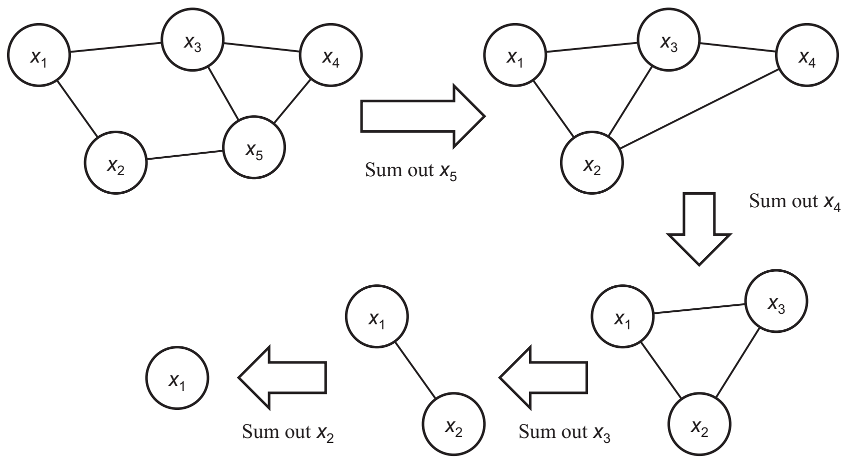
For this, we describe the elimination algorithm.

2 Visualizing the Elimination Algorithm

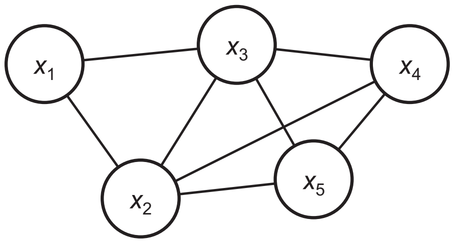
The Elimination Algorithm is for marginalizing out variables and boils down to just choosing an (elimination) ordering in which you want to sum out variables that you don't care about. Each time you sum out a variable, you push the summation into your factorization as much as possible so that you minimize the amount of work you have to do to sum out that variable. The bad news is that the elimination ordering can make a big difference and there is no known efficient (polynomial time) algorithm for determining the optimal elimination ordering given an arbitrary undirected graph.[1]

Graphically, what is going on each time you sum out a variable is that you connect all its neighbors into a clique (if they are not already a clique). For example, using elimination order $(5, 4, 3, 2, 1)$ in the following graph:

[1] In fact, finding the optimal elimination ordering is NP-hard, i.e., at least as hard as solving an NP-complete problem.



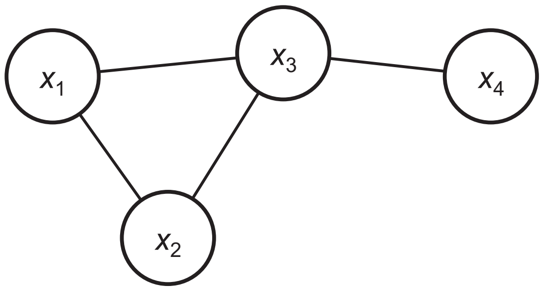
If you keep track of all the edges you added while doing this elimination procedure and read those to the original graph, you will end up with the reconstituted graph. For example, in the above diagram, note that the only time we ever introduced new edges was when we summed out x_5, during which we introduced edges (x_2, x_3) and (x_2, x_4). Adding these new edges to the original graph gives the reconstituted graph:



Reconstituted graph

3 Complexity of marginalization

Suppose we want to just sum out x_3 in the graph below:



The joint probability distribution factors as

$$p_{\mathsf{x}_1,\mathsf{x}_2,\mathsf{x}_3,\mathsf{x}_4}(x_1, x_2, x_3, x_4) \propto \varphi_{123}(x_1, x_2, x_3)\varphi_{34}(x_3, x_4).$$

In particular, we want to compute

$$p_{\mathsf{x}_1,\mathsf{x}_2,\mathsf{x}_4}(x_1, x_2, x_4) = \sum_{x_3} p_{\mathsf{x}_1,\mathsf{x}_2,\mathsf{x}_3,\mathsf{x}_4}(x_1, x_2, x_3, x_4) \propto \sum_{x_3} \varphi_{123}(x_1, x_2, x_3)\varphi_{34}(x_3, x_4).$$

What is the computational complexity of computing this? To foreshadow coding you will be doing later on in the class, consider the following pseudocode:

```
for each value of x_1:
    for each value of x_2:
        for each value of x_4:
            sum = 0
            for each value of x_3:
                sum = sum + phi_123(x_1, x_2, x_3)*phi_34(x_3, x_4)
            marginal(x_1, x_2, x_4) = sum
normalize marginal(x_1, x_2, x_4)
```

By counting the nested for-loops and the number of operations done at the inner-most loop, the overall computational complexity is $O(|\mathcal{X}|^4)$. Note that normalization takes $O(|\mathcal{X}|^3)$ operations since we need to sum over all the entries of the (unnormalized) marginal table to obtain the normalization constant Z and then divide every entry by Z.

4 Properties of Reconstituted graph

The reconstituted graph consists of the original graph together with all edges added during the elimination process. Formally, let $G = (V, E)$ be the original graph, and let E' be the set of edges added during elimination. Then, $G_{RC} = (V, E \cup E')$. Note thta the reconstituted graph can be different depending on the order chosen for elimination.

We now state 2 important properties of the reconstituted graph, along with the outlines of their proofs:

Theorem 1 *The size of the largest clique in the reconstituted graph determines the complexity of elimination.*
More precisely, let N^* be the maximum number of summations done at any step of the elimination algorithm, and let $\bar{C}$ be the maximum size of a clique in the reconstituted graph. Then $N^* \sim |\mathcal{X}|^{\bar{\mathcal{C}}}$.

Remark: The maximum size of the clique is not the same as size of the "maximal" clique. In fact, a graph may have many maximal cliques. We need to look at the largest clique to deterimine the complexity of elimination.

Proof:
Let $G = (V, E)$ be the original graph, and let G_{RC} be the reconstituted graph.

We need to prove 2 statements:

1) Suppose that the maximum number of variables summed in a single step of elimination is k. Then there exists a clique of size k in G_{RC}.
Proof: This direction is straightforward.

2) Suppose there exists a clique of size k in G_{RC}. Then there must have been a step in the elimination algorithm which involved summtion over k different variables.
Proof: This direction is slightly trickier. If the clique of size k existed in the original graph (G) itself, clearly we must have a summation over k variables during elimintaion. On the other hand, suppose this clique did not exist in G.
Let us consider how this clique may have been created. One possibility is that the first variable in this clique to be eliminated, say X_i, was connected to other (k-1) variables just before it was eliminated. In fact, this is the only way in which the clique could have been created! (Why?) The remainder of the proof follows directly from this observation.

Putting (1) and (2) together, we prove that the maximum number of variables involved in a summation step during elimination, equals the size of the largest clique in the reconstituted graph.
■

Theorem 2 *The reconsittuted graph obtained from elimination is a chordal graph. (Here, we assume that all the nodes of the graph, except possibly one, are eliminated.)*

Proof:
This is a simple proof by contradiction. Assume that the reconstituted graph is not chordal. Then it must have some cycle of length at least 4 with no shortcut. Can you use this property to derive a contradiction?
(We want you to finish the above proofs on your own)
■

Remark 1: You may contrast the chordal properties of a reconstituted graph with moralized graphs. Recall that a moralized graph is chordal if no edges are added during the moralization process. We may similarly say that an undirected graph is chordal, if no edges are added during the elimination process. However, note that the reconstituted graph is *always* chordal.

Remark 2: We saw that if elimination does not add any edges to a graph, then the graph must be chordal. You may also ask if the reverse is true - if the original graph is chordal, can we find an elimination order such that no edges will be added in the reconstituted graph? The answer is yes, and we will learn an efficient way to do this when we study Junction Trees.

Remark 3: Since we can do elimination on chordal graphs without adding edges, this means that the largest clique in the reconstituted graph is the same as the original graph. This means that the complexity is only $O(|\mathcal{X}|^{\mathcal{C}^*})$, where C^* is the largest clique in the original graph. This is unavoidable anyway, since specifying all the potentials would itself take $O(|\mathcal{X}|^{\mathcal{C}^*})$ time. This should convince you that chordal graphs are nice for doing inference!

5 Graph Triangulation

Triangulation[2] is the process of making graphs chordal by adding edges. Since all complete graphs are chordal, it is always possible to do this by adding enough edges! However, typically we will want to add as few edges as possible for making the graph chordal. This leads us to a well-known problem in computer science, viz. the minimum triangulation problem:

Minimum triangulation problem: Given an arbitrary graph G, find a graph G' over the same set of vertices, such that:
i) G is a subgraph of G'. (Or G' contains all the edges in G)
ii) G' is chordal.
iii) G' has the smallest number of edges, among all graphs satisfying (i) and (ii).

Let us now consider triangulation in a more important context viz. in the context of elimination. As we saw earlier, we can perform elimination on a chordal graph without adding any extra edges, and hence without increasing the maximum clique size. This gives us a nice procedure for doing inference - first triangulate the graph keeping the clique size as low as possible, and then run elimination with an ordering that doesn't add edges. This motivates the following optimal triangulation problem, also called the Tree-width problem:

Tree-width problem: Given an arbitrary graph G, find a graph G' over the same set of vertices, such that:
i) G is a subgraph of G'.
ii) G' is chordal.
iii) G' has the *smallest maximum clique size*, among all graphs satisfying (i) and (ii).

Unfortunately, both the above problems are known to be NP-hard! (To understand why this is such a disappointment, read the section on NP-hardness)

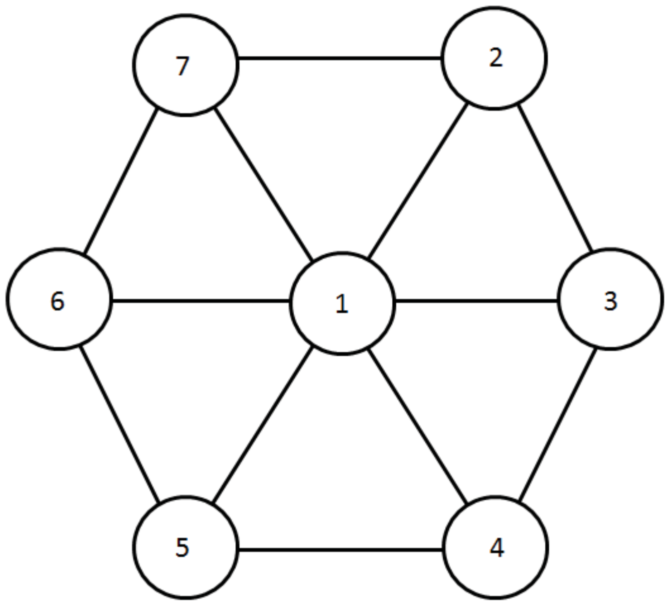
This points to a general trend about inference problems - they are usually NP-hard for general graphs, but become easy when restricted to chordal graphs. We will understand some of the reasons for the latter when we study junction trees.

Challenge: We had mentioned earlier that finding an optimal ordering for the elimination algorithm on a general graph is NP-hard. Above, we saw that the Tree-width problem is NP-hard. Prove that these 2 statements are equivalent.

[2]Be careful not to confuse this usage of the word ”triangulation” with another usage popular in computational geometry. In the latter, triangulation literally means dividing a (planar) graph into triangles.

Example of triangulation

Consider the following 7-node graph:



i) Is the above graph chordal? Why not?
ii) Triangulate the above graph. Try not to use too many edges!

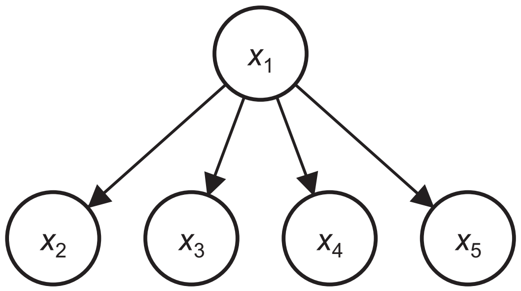
6 Elimination on directed graphical model

(important topic; not discussed in recitation)

Performing elimination directly on a directed graph is complicated. We will always use the following approach in practice:

Convert the directed graph into an undirected graph which is a minimal I-map; then use the standard elimination algorithm on the undirected graph. To do this, we will also need to convert the conditional probabilities for directed graph into potentials for the undirected graph, but you should convince yourself that this is easy. Here is an example problem to give you some practice with directed graphs:

Problem:
Consider the following DAG:



The corresponding probability distribution factors as:

$$p_{x_1,x_2,x_3,x_4,x_5} = p_{x_1} p_{x_2|x_1} p_{x_3|x_1} p_{x_4|x_1} p_{x_5|x_1}.$$

(a) Determine the moral graph.

(b) Determine the reconstituted graph for the worst elimination ordering. Note that the reconstituted graph is an undirected graph, so start with your answer to (a).
Remark: In general, there may be multiple worst elimination orderings.

(c) Let $\bar{x}_2$ be a constant. Simplify $p_{x_3|x_2}(x_3 \mid \bar{x}_2)$ as much as possible, writing it in terms of the given conditional probability tables (including p_{x_1}). Your answer may involve a proportionality.

7 A brief introduction to NP-hardness (optional reading)

NP-hardness is a recurring theme in this course. So, we give a brief introduction here, for those who do not have a computer science background. This section can be regarded as additional learning, and will not be important from point of view of the exams. However, it can be useful to improve your overall understanding of the course.

Q. What is polynomial time?
Ans. We say that an algorithm for a problem runs in polynomial time if its run-time on any input is bounded by some polynomial function of 'n', e.g. the run-time may be $O(n^2)$, or $O(n^3)$. Here 'n' represents the size of the problem, it can be thought of as the no. of bits used to describe the problem. Thus, in a factorization problem where the number given to you is N, the size of the problem is roughly log N, since we only need log N bits to describe a number of magnitude N.
In computer science, polynomial time has a strong connection with "efficiency" - a problem is said to be efficiently solvable if it can be solved in polynomial time in the size of the input.

Q. What is NP?
Ans. NP stands for Non-deterministic Polynomial Time, it is a class of problems that can be solved in polynomial time on a non-deterministic Turing machine. We won't go here

into the definition of Turing machines here. However, an equivalent way of characterizing NP-problems (which is very widely used in practice) is that it is a class of problems for which the solution can be verified quickly.

Q. What is NP-complete?
Ans. Informally, NP-complete problems are the 'hardest' problems in NP. Every other problem in NP can be reduced to an NP-complete problem, in polynomial time. In other words, giving a polynomial time algorithm for an NP-complete problem would make the entire class of problems NP solvable in polynomial time. In other words, we would have P = NP. The first NP-complete problem was SAT, short for Boolean Satisfiability problem, which was shown to be NP-complete by Cook in his 1971 paper.[3]
Most computer scientists believe P=NP to be false, and this gives us a strong reason to believe that NP-complete problems are "intrinsically hard" i.e. there does not exist any algorithm to solve them in polynomial time.

Q. What is NP-Hard?
Ans. This is a very similar notion to NP-completeness, except that an NP-hard problem itself need not be in NP. Thus, solving an NP-hard problem would prove P = NP, but the converse is not necessarily true.

Intuition: Let us start by considering the following scenario: Someone walks up to you and makes the following claim: I have checked all the numbers between 9,999 and 99,999 and there are exactly 1000 primes in this region. How would you verify whether this claim was true? The only way you could verify it is to check all the numbers between 9,999 and 99,999. Thus, checking the answer here requires you to solve the problem from scratch, or verification is *not* easier than solving.

Let's consider a variant of the above, where the person makes a different claim: "There is at least one prime number between 9,999 and 99,999." In this case, we can simply ask the person to tell us that number, say 85,679, and check whether this number is prime! [4] This is much easier than checking all the numbers in the given range. This is an example of a problem where verification is much easier than solving.

Example of an NP-complete problem: Subset-sum problem

Given a set of numbers, say: S = {1, -3, -5, 4, -9, -10, 5} We wish to determine if there exists a subset T of S such that the elements in T add up to zero. Of course, we could use the brute force method of checking all possible subsets of S, but that would take exponential time ($2^{|S|}$). Thus the question is whether there exists a way to solve this problem in polynomial-time.

[3]The paper is titled "The complexity of theorem-proving procedures"

[4]In fact, checking whether a number is prime can be done in polynomial time, as proved in the 2002 paper "Primes is in P"

Surprisingly, this simple-looking problem turns out to be NP-complete, which means there is probably no efficient way to solve it (if you believe P $\neq$ NP). However, if you are given a candidate solution, say R = {1, -3, 4}, you can quickly check whether:
i) The given set R is a subset of S.
ii) The elements of P add up to 0.
Thus, verification is easy. Hence, the problem is in NP. Showing that it is NP-complete, of course, takes much more work - we need to show that solving this problem efficiently gives a way to solve some NP-complete problem efficiently. We won't go into that here.

Example of an NP-hard problem that is not NP-complete: Maximum clique problem

In problem set-2, we showed that a graph can have exponentially many maximum cliques. Still, it may come as a surprise that finding the maximum clique size in a graph is NP-hard!

We can see why the problem is not in NP – even if I tell you what the largest clique in the graph is, you have no easy way to check that I am telling the truth. On the other hand, the decision version of this problem is NP-complete viz. "Does there exist a clique of size at least k?" This problem is in NP since a solution to such a problem is easy to check. You can ask me to provide you the description of the size $\geq$k clique, and you can quickly check that it is a valid clique, with size at least k

(Remember that $NP - Hard \cap NP \iff NP - complete$)

Example of a 'hard' problem that is not NP-hard: Factorization

Given a positive integer N, output its prime factors $p_1, p_2, \ldots p_k$, as well as the exponents of each prime factor, $\alpha_1, \alpha_2 \ldots \alpha_k$, in some pre-determined format. This problem is in NP because given a prime factorization, it is easy to check whether the factors are really prime and whether their product equals N. On the other hand, it is not an NP-complete problem, which means that it could have a polynomial time solution even if $P \neq NP$. However, most mathematicians and computer scientists still believe this problem to be hard. In fact, a lot of our cryptographic systems rest on the hardness of this problem.

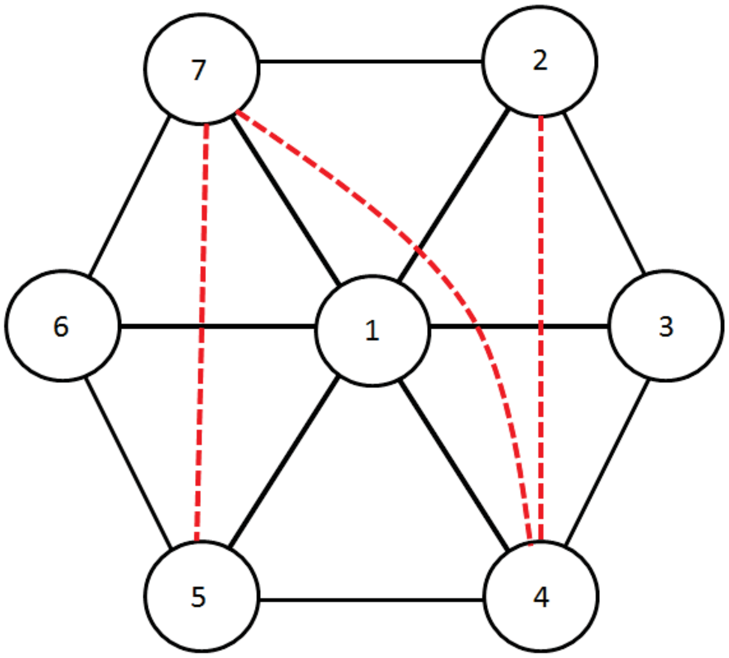
Summary:

NP-hardness gives theorists a way of "proving" that some problem is intrinsically hard, so that we stop searching for an efficient solution to it. This is important because of 2 reasons:
1) A lot of practically important problems turn out to be NP-hard.
2) We don't have any other useful notion for proving hardness of a problem.

We will also prove that a lot of probabilistic inference problems are hard on general graphs, and this will justify our restriction to special types of graphs (e.g. chordal graphs) in this course.

8 Solutions

8.1 Solution to triangulation problem:



Of course, there can be many other symmetric solutions with 3 edges.