CMPSCI 611 Advanced Algorithms **Fall 2000**

Lecture 20: November 21

Lecturer: Micah Adler *Scribe: Ping Ji*

We know that obtaining an optimal solution for an **NP–complete** problem is intractable, but, if we relax the definition of *solving* for the **NP–complete** problems, it is not impossible to get satisfiable solutions. There are two common ways of getting around **NP–completeness**. The first is to look at some restrictions on the problem that allow us to find sufficient solutions, and the second is to look for a nearly-optimal solution instead of an optimal one. This approach will lead to using approximation algorithms. This lecture addresses three strategies for tackling **NP–complete** problems, among which the approximation algorithm strategy is the one that we are most interested in.

20.1 Restrict The Allowed Input

If we restrict the type of input allowed some **NP–complete** problems can be solved in polynomial time.

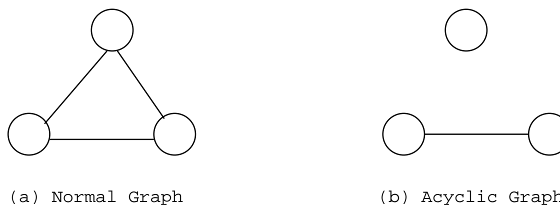
20.1.1 Satisfiability Problem

2-SAT is a restriction of the satisfiability problem. The satisfiability problem allows any kind of boolean formula, and 2-SAT only allows 2-CNF formulae, obviously a restriction, and we've already proved that 2-SAT problem can be solved in polynomial time. Thus by restricting the input of a satisfiability problem to 2-CNF, it can be solved in polynomial time. This is a typical example of the strategy of restricting the allowed input to reduce the complexity of a problem.

20.1.2 Graph Problems

Many graph problems are **NP–complete**. But if we put constraints on the type of the input graph, we could solve some graph problem more easily. For example, for the well known CLIQUE problem, if an input graph of CLIQUE problem is one of the following graphs, CLIQUE can be solved in polynomial time.

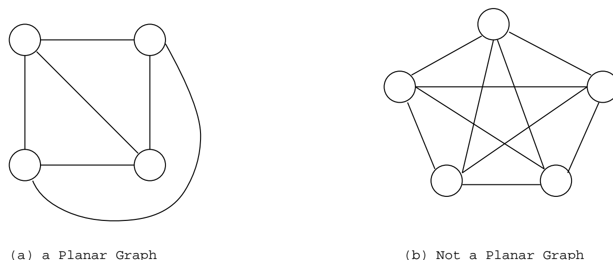
- *Acyclic graph:* a graph without cycles. In such a graph we can not even have a triangle, see Figure 20.1. If the input to the CLIQUE problem is an acyclic one we can solve it in polynomial time.



(a) Normal Graph (b) Acyclic Graph

Figure 20.1: Acyclic Graph

- *Bound degree:* a graph all of whose vertices have degree at most r, for some constant r. If the vertices of an input graph have an upper bound $(r-1)$ for their degree, the largest CLIQUE that we can have is of size r. Since r is constant, we can test all of the possible cliques in time $O(n^r)$, which is polynomial solvable. So if we have a constant degree-bound on a graph, we can solve CLIQUE and its related graph problems in polynomial time.
- *Planar graph:* a graph that can be drawn in 2-dimensions without crossing edges. For example, a complete graph with four vertices is a planar graph, since it can be drawn in the way that is shown in Figure 20.2(a). On the other hand, a complete graph of five vertices is not a planar graph, since we can not find a way to draw it without crossing edges (Figure 20.2(b)). Again, if the input graph is a planar graph, we can solve CLIQUE very easily.



(a) a Planar Graph (b) Not a Planar Graph

Figure 20.2: Planar Graph

- *Chordal graph:* If a graph has any cycle of length 4 or more, then there is at least one inner cycle edge present. Figure 20.3 shows an example of a chordal graph. If the input graph is a chordal graph, we can solve the clique problem on it in polynomial time.

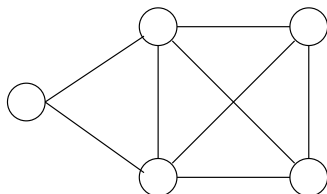


Figure 20.3: Chordal Graph

20.1.3 Constraint on Input Integer

Another strategy used is that we might assume integers of the input are polynomial in input size. For some of the problems, if we don't make assumptions for the length of the input, the size of the input integer could be exponential. On the other hand, if we make assumptions on the size of the input integer, some **NP–complete** problem can be solved more easily. For example, the Knapsack problem can be done in polynomial time if the input "total weight" is of polynomial size.

If we assume that the input integers are polynomial size, some **NP–complete** problems can be solved in polynomial time, but some can not. Because of this, people have developed terminologies to differentiate the problems having this property and the problems without this property.

Definition 20.1 *An algorithm runs in* Pseudo-polynomial time *if the running time is polynomial in input size and any integer in the input.*

Example 1: For the Knapsack problem we have a pseudo-polynomial time algorithm (using dynamic programming) to solve it.

Example 2: The Subset-Sum problem is an **NP–complete** problem. What the Subset-Sum problem asks is whether there is a subset of the given set of integers such that the sum of the subset is equal to a predefined value. We can solve the Subset-Sum problem in Pseudo-polynomial time by using a dynamic programming algorithm similar to the Knapsack solutions.

Example 3: The Partition problem, which is very closely related to the Subset-Sum problem. In the Partition problem, we are given a list of integers, and we need to answer whether it can be divided into two subsets that both sum to the same value. In another word, if we sum all the integers in the input set and divide that value by 2, we are actually asking whether there is a subset-sum to this value. Both the Subset-Sum and Partition problems are **NP–complete**, but they can be solved in Pseudo-polynomial time.

In the meantime, for the CLIQUE problem the only integer input is the size of the CLIQUE (k). We don't need to put any constraint on k to reduce the complexity, because it is already polynomial. In other words, we can not solve the CLIQUE problem in Pseudo-polynomial time. We'll give another definition for problems that can not be solved in Pseudo-polynomial time.

Definition 20.2 *A problem is **Strongly** **NP–complete** if it remains **NP–complete** even when we restrict all the integers in an input of length n to be polynomial in n.*

- **Example:**The Bin-Pack problem.

  Input: A set of items of integer size, a bin size b, and an integer k.

  Question: can we partition the items into at most k sets (bins) such that no bin has total size larger than b (i.e., the size of each set $\leq b$)?

We know that a problem is either one that can be solved in Pseudo-polynomial time or it is one that is **Strongly NP–complete**. No problem can be both. This can be shown by assuming that the Knapsack problem were Strongly **NP–complete**, then the dynamic programming algorithm would actually prove **P** = **NP**. If the Knapsack problem remained **NP–complete** even with the restriction on the size of its input integer, we would have an algorithm that could solve all the problems in **NP** in polynomial time, which gives us **P** = **NP**. Therefore, a problem can NOT be both Pseudo-polynomial solvable and **Strongly NP–complete**.

20.2 Assuming Probability Distribution Over the Input

The second strategy that we can use to solve **NP–complete** problems is to assume that the input follows some kind of probability distribution.

The *good news* about this strategy is that many **NP–complete** problems will be solvable in the average case (e.g., the Hamiltonian path problem). The Hamiltonian path problem on a random input is solvable in polynomial time. (Random input means choose a random graph where every edge is present with a certain probability.) With this kind of input we can show, with high probability, whether a graph has a Hamiltonian path in polynomial time.

The *bad news* about this strategy is twofold. Firstly, the worst case might be a very important one in many practical problems. This strategy might work badly on the worst case, and that may cause trouble in real applications. Secondly, it is not easy to figure out clearly what is the correct probability distribution.

An interesting question is "can **NP–complete** problems be solved by using randomized algorithms?". Well, it turns out that if one of the **NP–complete** problems can be solved by a randomized algorithm then all of the **NP** problems can be solved by using randomized algorithms, which people generally believe is NOT true!

20.3 Approximation Algorithm

Now, let's turn to the strategy that we are most interested in – using approximation algorithms! The approximation algorithm is usually the best solution for solving **NP–complete** problems. By using an approximation algorithm, we are going to find a *nearly* optimal solution and give up the optimal one. Let's see an example first.

20.3.1 Using Greedy Algorithm for Vertex Cover:

The Vertex Cover problem:
Input: An undirected graph G=(V,E).
Output: A set of vertices $U \subseteq V$ of minimum size, such that $\forall e \in E$, e has at least one endpoint in U.

The Vertex Cover problem has been proved to be **NP–complete**. For it's decision version, the approximation algorithm doesn't make sense. But for the min-value Vertex Cover problem, the approximation algorithm does help. In fact for the min-value Vertex Cover we want to cover as many edges using the minimum number of vertices or cut off the maximum degree edges. A greedy algorithm is a natural choice for this. But greedy algorithms turns out NOT to be a very good approximation scheme for the min value version of Vertex Cover. (Actually for many **NP–complete** problem, a greedy algorithm can not offer a good approximation solution.) Anyway, let's first take a look at how a greedy algorithm works for finding a *nearly* optimal solution for Vertex Cover.

1. $U = \emptyset$
2. while $E \neq \emptyset$ do
3. let u be the maximum degree vertex
4. $U = U + u$;
5. $G = G - u$; return U; //remove u and its corresponding edges from graph G 6.

By using this greedy algorithm, the approximation solution of min-value Vertex Cover for Figure 20.4(a) is shown in Figure 20.4(b), and the optimal solution is shown in Figure 20.4(c).

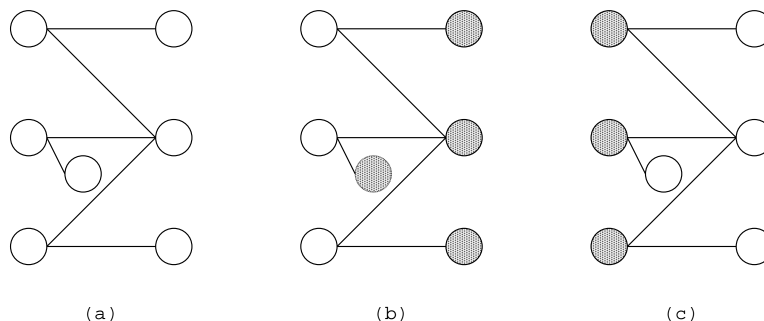


Figure 20.4: Comparison of Greedy Solution and Optimal Solution.

The solution found by using the greedy algorithm can be off from (or can be worse than) the optimal solution by factor of $\Theta(\log(|V|))$. We can show it by giving a specific graph whose solution is off from the optimal by this factor.

Figure 20.5 shows the graph that we use. In this graph, each column has the number of vertices shown in the first row of Figure 20.5. The first column has m vertices (which is 6 in this example), the second column has $\lfloor \frac{m}{2} \rfloor$ vertices, the third column has $\lfloor \frac{m}{3} \rfloor$, ..., and the k-th column has $\lfloor \frac{m}{k} \rfloor$ vertices, where $k = 1, 2, ..., m$. At the same time, every vertex in the k-th ($k > 1$) column will have k edges connecting to k vertices of the first column, which means that except the first column, the degree of a vertex in the k-th column equals k. And for different vertices in the k-th ($k > 1$) column, the sets of vertices in the first column that they connect to are disjoint. Each of the vertices in the second column connects to 2 vertices in the first column, and the set of vertices connecting to the first vertex in column 2 does not overlap with the set of vertices connecting to the second vertex in column 2, and so on. Also notice that the vertex in the first column has degree $\leq (m - 1)$.

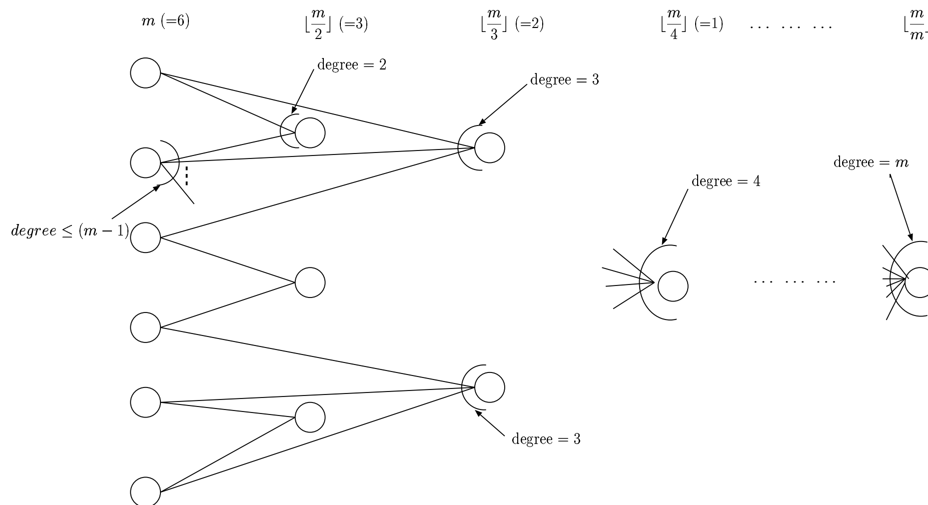


Figure 20.5: Greedy Algorithm for Min-Value Vertex Cover

Given this specific graph the proposed greedy algorithm will remove the right most column of vertices in the graph since, at the beginning, the vertex in the last column has the biggest degree ($= m$), and after it is removed the vertices in the first column will have degree at most $(m - 2)$, which is not bigger than the degree of vertices in the $(m - 1)$-th column, etc.. We are now interested in how well the approximation-algorithm (the greedy algorithm) does for this particular input. Let's compare the optimal solution and the approximation solution.

Optimal solution: m. If we take all the vertices in the first column, every edge in this input graph will be covered. Thus the optimal solution is m.

The greedy algorithm: $\Theta(m \log m)$. Using the greedy algorithm the vertices that we choose will be all those to the right of the first column. Therefore the total number of vertices that we get for the Vertex Cover

problem using the greedy algorithm is:

$$\begin{aligned}
\lfloor\frac{m}{2}\rfloor + \lfloor\frac{m}{3}\rfloor + ... + 1 &= (m + \lfloor\frac{m}{2}\rfloor + \lfloor\frac{m}{3}\rfloor + ... + 1) - m \\
&= (m + 1 + \lfloor m2\rfloor + 1 + \lfloor\frac{m}{3}\rfloor + 1 + ... + 1 + 1) - 2m \\
&\geq m(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... + \frac{1}{m}) - 2m \\
&= \Theta(m \log m).
\end{aligned}$$

20.3.2 Performance Ratio for Approximation Algorithms

For quantifying the performance of an approximation algorithm, we define a quantity named the *performance ratio*. Actually, the performance ratio depends on whether the problem is a maximization problem or a minimization problem.

Definition 20.3 *For the minimization problem, the performance ratio is*

$$\max_{x,|x|=n} \left(\frac{C_{alg}(x)}{C_{opt}(x)}\right) \tag{20.1}$$

Where $C_{alg}(x)$ is the cost of approximation solution on input x; and $C_{opt}(x)$ is the cost of the optimal solution on input x.

The basic ratio is a function of n, and we could have a different value for every possible input of size n. Thus we choose the performance ratio to be the maximum, over all inputs of size n, of the ratio of how well the algorithm does to the optimal cost. The performance ratio for minimization problems is at least 1.

Example: For the greedy algorithm for the Vertex Cover problem, the performance ratio can be calculated as:

$$\Omega\left(\frac{m \log m}{m}\right) = \Omega(\log m)$$

Here, m is not the input size! In fact, if there are m vertices in the first column of Figure 20.5, then the input size is $\Theta(m^2)$. Even though the performance ratio of the greedy algorithm that we used for Vertex Cover is still $\Omega(\log m)$, that is because we are using the log function on the input size of $\Theta(m^2)$.

Definition 20.4 *For the maximization problem, the performance ratio is*

$$\max_{x,|x|=n} \left(\frac{C_{opt}(x)}{C_{alg}(x)}\right) \tag{20.2}$$

Where $C_{alg}(x)$ is the cost of approximation solution on input x; and $C_{opt}(x)$ is the cost of the optimal solution on input x.

The performance ratio for maximization problems is the maximum, over all inputs of size n, of the ratio of the optimal cost to the algorithm's cost. For maximization problems, the performance ratio is at most 1.

From definitions 20.3 and 20.4, we say that an $f(n)$-*approximation* is an approximation algorithm with performance ratio $f(n)$. Furthermore, we have an alternative measurement that is less commonly used for the performance of approximation algorithm is:

$$\frac{|C_{alg} - C_{opt}|}{C_{opt}} \tag{20.3}$$

20.3.3 Better Approximation for Vertex Cover:

We have a better approximation algorithm than the greedy algorithm for the Vertex Cover problem. It is the best known approximation algorithm for Vertex Cover. It can provide a much better performance ratio than Greedy algorithm does.

1. $S = \emptyset$;
2. while $E \neq \emptyset$ do
3. pick any edge $e = (u, v)$
4. $S = S + u + v$;
5. $G = G - u - v$;
6. return S;

Claim 20.5 *The performance ratio of this algorithm is 2, or, this algorithm is a 2-approximation.*

Proof:

1. Let E' denotes the edges chosen by the algorithm. Then the cost of the algorithm is:

$$C_{alg}(G) = 2 \cdot |E'|$$

2. We observe that E' is a matching, because whenever we put a vertex into set S, we remove all the edges that are incident to it. Thus, we can only choose one edge incident to any vertex during the algorithm. Therefore E' is a matching.

3. $\Rightarrow C_{opt}(G) \geq |E'|$. Since, if you have a matching, all the edges have to have at least one vertex in the optimal solution, and there is no overlapping of the vertices because of the matching.

4. 1. and 3. $\Rightarrow$ For any G, $C_{alg}(G) \leq 2 \cdot C_{opt}(G)$.

5. We can show that $C_{alg}(G) = 2 \cdot C_{opt}(G)$ for the worst cast input. The worst case input is actually a graph consisting of a matching. Using the above algorithm every edge of the input graph (the matching) has to be chosen, which means that we need to put all of the vertices into set S. But the optimal solution to this case is only half of the size of total vertices.

6. 4. and 5. $\Rightarrow$ the performance ratio of this algorithm= 2.

■

20.3.4 Solving Independent-Set Problem

If we can approximate one **NP–complete** problem, can we approximate every **NP–complete** Problem? The answer is **NO**!. Something that people might try is to reduce an **NP–complete** problem to the approximated **NP–complete** problem (say the approximation solution of Vertex Cover), use the approximation solution and then reduce it back to the original **NP** problem that we want to solve. This turns out **NOT** to work!

Let's look at an example: solving the Independent-Set problem:

Input: An undirected graph $G = (V, E)$.

Output: A set $U \subseteq V$ of maximum size such that no two vertices in U are connected by an edge.

Claim 20.6 *Independent-Set $\leq_p$ Vertex Cover.*

Proof: We know that if U is an Independent-Set then $V-U$ is a Vertex Cover. We can do the following reduction:

1. $G \rightarrow G$;
2. $k \rightarrow |V|-k$;

This reduction can be completed in polynomial time. We can show that this reduction is correct:

$$\begin{aligned}\text{U is Independent-set} &\leftrightarrow \text{No edge has both endpoints in the set U}\\ &\leftrightarrow \text{every edge has at least one end point in set } V-U\\ &\leftrightarrow V-U \text{ is a Vertex Cover}\end{aligned}$$

Therefore, we can reduce Independent-Set to Vertex Cover in polynomial time. ■

"Approximation" for the Independent-Set Problem:

1. Find S: the approximation set for Vertex Cover; //using the approximation algorithm in section 20.3.3
2. Return $V-S$.

This *"approximation"* algorithm is not efficient for solving Independent-Set problem. We can see this from Figure 20.6. For the input graph of Figure 20.6(a), we have the optimal Independent-Set as shown in Figure 20.6(b). But by running the above approximation algorithm, we will get the very bad solution shown in Figure 20.6(c).

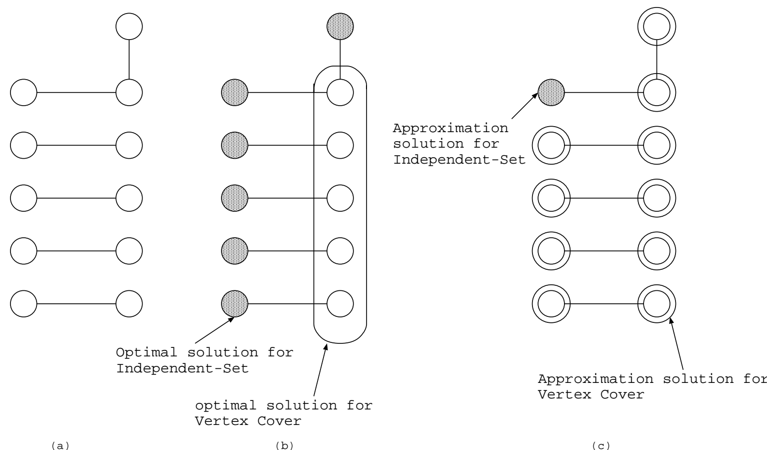


Figure 20.6: Independent-Set Problem

The reason that this scheme does not work well is that when we do the reduction, we only preserve the decision version of the problem, which has nothing to do with the quantity solution and the performance ratio. Therefore, a good performance ratio of an approximation solution can not be guaranteed if reduction is used. During the next several lectures, we are going to talk about better approximation algorithms used to solve **NP–complete** problems.