Discrete Mathematics and Algorithms
ICME Refresher Course

Nolan Skochdopole

September 17, 2015

These notes are adapted from last year's notes, written by Austin Benson (also on website for the refresher course).

These are meant to be a preparation of sorts for CME 305 (Discrete Mathematics and Algorithms) in the winter quarter. The class can be done without many prerequisites as most everything is taught from the basics, but the pace can be tough for people who have never seen some the stuff in the class before. So I want to use this course to expose you to many of the topics you will see in much more depth during the class just so you will have seen it before, especially all the definitions for basic graph theory.

The textbook the class recommends is *Algorithm Design* by Kleinberg and Tardos. The instructor also recommends *Graph Theory* by Dietsel as a refresher for graph theory.

1 Computational complexity and big-O notation

References: [Ros11]

The time that algorithms take to solve problems depends on the implementation, the software, the hardware, and a whole host of factors. We use *big-O* notation as a way of simplifying the running time of an algorithm based on the size of its input. The notation allows us to talk about algorithms at a higher level and estimate how implementations of algorithms will behave. We will use big-O notation extensively in this refresher course.

Definition 1.1. Let f, g: $\mathbb{R} \to \mathbb{R}$. We say that f is $O(g)$ if there are positive constants c and N such that for $x > N$, $|f(x)| \leq c|g(x)|$.

Example 1.2. *The number of multiplications and additions needed to multiply two $n \times n$ matrices together is $n^2(2n-1)$, which is $O(n^3)$.*

To see Example 1.2, let $f(n) = n^2(2n-1)$ and $g(n) = n^3$. Choose $N = 1$ and $c = 2$. For $n > N$, $f(n) = 2n^3 - n^2 < 2n^3 = cg(n)$. A general advantage of big-O notation is that we can ignore the "lower order terms":

Example 1.3. *Let a_0, a_1, $\ldots$, a_k be real numbers. Then $f(x) = a_0 + a_1x + \ldots a_kx^k$ is $O(x^k)$. To see this, choose $c = |a_0| + |a_1| + \ldots + |a_k|$ and $N = 1$.*

Exercise 1.4. *Show the following:*

- *If f_1 and f_2 are both $O(g)$, then so are $\max(f_1, f_2)$ and $f_1 + f_2$.*
- *If f_1 is $O(g_1)$ and f_2 is $O(g_2)$, then $f_1 f_2$ is $O(g_1 g_2)$.*
- *$\log n!$ is $O(n \log n)$.*

Theorem 1.5. *For any k, x^k is $O(2^x)$.*

Proof. The taylor series of 2^x about $x = 0$ is

$$2^x = 1 + (\ln 2)x + \frac{\ln^2 2}{2}x^2 + \ldots + \frac{\ln^k 2}{k!}x^k + \frac{\ln^{k+1} 2}{(k+1)!}x^{k+1} + \ldots$$

Choose $c = \frac{\ln^k 2}{k!}$ and $N = 1$. □

The big-O notation allows us to say that some functions are smaller than other functions in an asymptotic sense. In other words, big-O notation lets us describe *upper bounds*. We may also want to describe whether or not upper bounds are tight, lower bounds, and asymptotic equivalency. There is an alphabet soup that lets us describe these relationships.

Definition 1.6. Let f, g: $\mathbb{R} \to \mathbb{R}$.

- Ω (**big-Omega**): We say that $f(x)$ is $\Omega(g(x))$ if there exists positive constants c and N such that for $x > N$, $|f(x)| \geq c|g(x)|$.
- Θ (**big-Theta**): We say that $f(x)$ is $\Theta(g(x))$ if there exists positive constants c_1, c_2, and N such that for $x > N$ $c_1 g(x) \leq f(x) \leq c_2 g(x)$. Equivalently, $f(x)$ is $O(g(x))$ and $\Omega(g(x))$.
- o (**little-o**): We say that $f(x)$ is $o(g(x))$ if for every positive constant ϵ, there is a positive constant N such that for $x > N$, $|f(x)| \leq \epsilon|g(x)|$.
- ω (**little-omega**): We say that $f(x)$ is $\omega(g(x))$ if for every positive constant C, there is a positive constant N such that for $x > N$, $|f(x)| \geq C|g(x)|$.

Wikipedia has a nice summary of this notation at http://en.wikipedia.org/wiki/Asymptotic_notation#Family_of_Bachmann.E2.80.93Landau_notations.

Example 1.7. *We have that*

- *$\cos(x) + 2$ is $\Omega(1)$.*
- *$n^2(2n - 1)$ is $\Theta(n^3)$.*
- *$n \log n$ is $o(n^2)$.*
- *For any $\epsilon > 0$, $\log n$ is $o(n^\epsilon)$.*
- *n^2 is $\omega(n \log n)$.*

Theorem 1.8. *$\log n!$ is $\Theta(n \log n)$*

Proof. In Exercise 1.4 we shows that $\log n!$ is $O(n \log n)$ so it is sufficient to show that $\log n!$ is $\Omega(n \log n)$. Note that the first $n/2$ terms in $n! = n \cdot (n-1) \cdot \ldots \cdot 1$ are all greater than $n/2$. Thus, $n! \geq (n/2)^{n/2}$. So $\log n! \geq \log{(n/2)^{n/2}} = (n/2)\log(n/2) = (n/2)(\log n - \log 2)$, which is $\Omega(n \log n)$. □

Theorem 1.9. *Let a_0, a_1, …, a_k be real numbers with $a_k > 0$. Then $f(x) = a_0 + a_1 x + \ldots a_k x^k$ is $\Theta(x^k)$.*

Proof. From Example 1.3, $f(x)$ is $O(x^k)$. Let $d = \max_{1 \leq i \leq k-1} |a_i|$. Then

$$\begin{aligned} |f(x)| &= \left|a_0 + a_1 x + \ldots + a_k x^k\right| \\ &\geq a_k x^k - |a_{k-1}|x^{k-1} - |a_{k-2}|x^{k-2} - \ldots - |a_0| \\ &\geq a_k x^k - d x^{k-1} \text{ for } x > 1 \end{aligned}$$

Let $c = a_k/2$ Thus, $|f(x)| \geq c|x^k|$ if $(a_k/2)x^k - dx^{k-1} \geq 0$, which holds for $x > 2d/a_k$. Hence, $|f(x)| \geq c|x^k|$ for all $x > \max(1, 2d/a_k)$. □

Theorem 1.10. *$\binom{n}{k}$ is $\Theta(n^k)$ for fixed constant $k \leq n/2$.*

Proof. Note that

$$\binom{n}{k} = \frac{n(n-1)\ldots(n-k+1)}{k!}$$

This is just a polynomial with leading term n^k, so $\binom{n}{k}$ is $\Theta(n^k)$ by Theorem 1.9. □

2 Recurrence relations

References: [Ros11, DPV06]

2.1 Definitions and simple examples

Definition 2.1. A recurrence relation for a function $T(n)$ is an equation for $T(n)$ in terms of $T(0)$, $T(1)$, ..., $T(n-1)$.

Example 2.2. *A well-known recurrence relation is the Fibonacci sequence:*

- $f_0 = 0,\ f_1 = 1$
- $f_n = f_{n-1} + f_{n-2}, \quad n \geq 2$

Recurrence relations are often the easiest way to describe a function, and there are a few reasons why we are interested in them. First, we can solve recurrence relations to get explicit formulae for functions. Second, we can use recurrence relations to analyze the complexity of algorithms.

Theorem 2.3. *Suppose we have an equation $x^2 - ax - b = 0$ with distinct roots r and s. Then the recurrence $T(n) = aT(n-1) + bT(n-2)$ satisfies*

$$T(n) = cr^n + ds^n,$$

where c and d are constants.

Typically, the constants c and d can be derived from the base cases of the recurrence relation. We can now find an explicit formula for the Fibonacci sequence. Set $a = b = 1$, so the equation of interest is

$$x^2 - x - 1 = 0,$$

which has roots $\left(1 \pm \sqrt{5}\right)/2$. By Theorem 2.3,

$$f_n = c\left[\left(1+\sqrt{5}\right)/2\right]^n + d\left[\left(1-\sqrt{5}\right)/2\right]^n$$

Plugging in $f_0 = 0$ and $f_1 = 1$ gives $c = 1/\sqrt{5}$ and $d = -1/\sqrt{5}$. Therefore,

$$f_n = \frac{1}{\sqrt{5}}\left[\left(1+\sqrt{5}\right)/2\right]^n - \frac{1}{\sqrt{5}}\left[\left(1-\sqrt{5}\right)/2\right]^n$$

Theorem 2.3 assumed that the equation had two distinct roots.

Theorem 2.4. *Suppose that the equation $x^2 - ax - b = 0$ has one (repeated) root r. Then the recurrence $T(n) = aT(n-1) + bT(n-2)$ satisfies*

$$T(n) = cr^n + dnr^n,$$

where c and d are constants.

Example 2.5. *Consider the following game. In the first two steps of the game, you are given numbers z_0 and z_1. At each subsequent step of the game, you flip a coin. If the coin is heads, your new score is four times your score from the previous step. If the coin is tails, your new score is the negation of two times your score from the two steps ago. What is your expected score at the n-th step of the game?*

Let X_n be the score at the n-th step of the game.

$$X_n = 4X_{n-1}\mathbb{I}\left(n\text{-th coin toss is heads}\right) - 2X_{n-2}\mathbb{I}\left(n\text{-th coin toss is tails}\right)$$

By linearity of expectation and independence of the n-th coin toss from the previous scores,

$$\mathbb{E}[X_n] = 4\mathbb{E}[X_{n-1}]\Pr\left[\text{heads}\right] - 2X_{n-2}\Pr\left[\text{tails}\right] = 2\mathbb{E}[X_{n-1}] - \mathbb{E}[X_{n-2}]$$

Let $T(n) = \mathbb{E}[X_n]$. By Theorem 2.4, we are interested in the roots of the equation $x^2 - 2x + 1 = 0$, which has a single repeated root $r = 1$. Thus, $T(n) = c1^n + dn1^n = c + dn$. Plugging in $T(0) = z_0$ and $T(1) = z_1$, we get $c = z_0$ and $d = z_1 - z_0$. Therefore, the expected score at the n-th step is

$$z_0 + (z_1 - z_0)n.$$

2.2 The master theorem and examples

Now we will switch gears from getting exact functions for recurrences to analyzing the asymptotic complexity of algorithms. The theory is summarized in Theorem 2.6.

Theorem 2.6. ***The master theorem.*** *Suppose that $T(n) = aT\left(\lceil n/b\rceil\right) + O(n^d)$ for constants $a > 0$, $b > 1$, $d \geq 0$, then*

$$T(n) = \begin{cases} O(n^d) & d > \log_b a \\ O(n^d \log n) & d = \log_b a \\ O(n^{\log_b a}) & d < \log_b a \end{cases}$$

2.2.1 Fast matrix multiplication

The first example we are going to work through is fast matrix multiplication. Consider multiplying $C = A \cdot B$, where A and B are both $n \times n$ matrices with n a power of two:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix},$$

where we have partitioned the matrices into four sub-matrices. Multiplication with the classical algorithm proceeds by combining a set of eight matrix multiplications with four matrix additions:

$$\begin{aligned} M_1 &= A_{11} \cdot B_{11} & M_2 &= A_{12} \cdot B_{21} & M_3 &= A_{11} \cdot B_{12} & M_4 &= A_{12} \cdot B_{22} \\ M_5 &= A_{21} \cdot B_{11} & M_6 &= A_{22} \cdot B_{21} & M_7 &= A_{21} \cdot B_{12} & M_8 &= A_{22} \cdot B_{22} \end{aligned}$$

$$\begin{aligned} C_{11} &= M_1 + M_2 & C_{12} &= M_3 + M_4 \\ C_{21} &= M_5 + M_6 & C_{22} &= M_7 + M_8 \end{aligned}$$

Let $T(n)$ be the number of floating point operations used to multiply two $n \times n$ matrices with this algorithm. There are two costs: the eight multiplications and the four additions. Since adding two $n \times n$ matrices uses $O(n^2)$ floating point operations, the cost of the algorithm is

$$T(n) = 8T(n/2) + O(n^2)$$

Applying the master theorem with $a = 8$, $b = 2$, and $d = 2$, we get that

$$T(n) = O\left(n^{\log_2 8}\right) = O(n^3).$$

Can we do better with a recursive algorithm? Looking at Theorem 2.6, if we could reduce the number multiplications and increase the number of additions, we could get a faster algorithm. This is exactly the idea of Strassen's algorithm:

$$\begin{array}{llll} S_1 = A_{11} + A_{22} & S_2 = A_{21} + A_{22} & S_3 = A_{11} & S_4 = A_{22} \\ S_5 = A_{11} + A_{12} & S_6 = A_{21} - A_{11} & S_7 = A_{12} - A_{22} & \\ T_1 = B_{11} + B_{22} & T_2 = B_{11} & T_3 = B_{12} - B_{22} & T_4 = B_{21} - B_{11} \\ T_5 = B_{22} & T_6 = B_{11} + B_{12} & T_7 = B_{21} + B_{22} & \end{array}$$

$$M_r = S_r T_r, \quad 1 \le r \le 7$$

$$\begin{array}{ll} C_{11} = M_1 + M_4 - M_5 + M_7 & C_{12} = M_3 + M_5 \\ C_{21} = M_2 + M_4 & C_{22} = M_1 - M_2 + M_3 + M_6 \end{array}$$

This algorithm uses seven matrix multiplications multiplications (the M_r) and 18 matrix additions / subtractions. The running time is thus

$$T(n) = 7T(n/2) + O(n^2)$$

Applying Theorem 2.6 gives $T(n) = O(n^{\log_2 7}) = o(n^{2.81})$.

2.2.2 k-select

Our second application of the master theorem is the analysis of the k-select algorithm (Algorithm 1). The k-select algorithm finds the k-th smallest number in a list of numbers L. The algorithm can be used to find the median of a set of numbers (how?). This algorithm is similar to Quicksort, which we will see in Section 3.

We are interested in the running time of k-select algorithm. Note that we can form the lists $L_<$, $L_>$, and $L_=$ in $\Theta(n)$ time, where n is the number of elements in the list L. It might happen that the pivot element is always chosen to be the largest element in the array, and if $k = 1$, then the algorithm would take $n + (n-1) + \ldots 1 = \Theta(n^2)$ time. While the *worst-case* running time is $\Theta(n^2)$, we are also interested in the *average-case* running time.

Data: list of numbers L, integer k
Result: the k-th smallest number in L, x_*
pivot $\leftarrow$ uniform random element of L
$L_< \leftarrow \{x \in L | x < pivot\}$
$L_> \leftarrow \{x \in L | x > pivot\}$
$L_= \leftarrow \{x \in L | x = pivot\}$
if $k \leq |L_<|$ **then**
| $x_* \leftarrow$ `select(`$L_<$`, `k`)`
end
else if $k > |S_<| + |S_=|$ **then**
| $x_* \leftarrow$ `select(`$L_>$`, `$k - |S_<| - |S_=|$`)`
end
else
| $x_* \leftarrow$ pivot
| return
end

Algorithm 1: k-select algorithm, `select(L, k)`

Let's consider how long it takes, on average, for the current list to have $3n/4$ or fewer elements. This occurs if we choose a pivot with at least $1/4$ of the elements of L smaller and at least $1/4$ of the elements are larger. In other words, the pivot has to be in the middle half of the elements of L. Since we choose the pivot uniformly at random, this condition holds with probability $1/2$ in the first call to the k-select algorithm. The expected number of steps until choosing a pivot in the middle half of the data is two (you should be able to solve this after the first probability/statistics refresher lecture).
The *expected* running time of the algorithm, $T(n)$ satisfies

$$T(n) \leq T(3n/4) + O(n)$$

The $O(n)$ term comes from the two steps needed (in expectation) to find a pivot that partitions the data so that the algorithm operates on at most $3/4$ the size of the original list. Using Theorem 2.6, the expected running time is $O(n)$. Since we have to read all n elements of L, the expected running time is $\Theta(n)$.

3 Sorting

References: [CSRL01, DPV06]

The k-select algorithm gave us a taste for ordering of a list of numbers. In general, the task of ordering all elements in some set is called *sorting*.

3.1 Insertion sort

Insertion sort works by inserting an element into an already sorted array. Starting with a sorted list of n numbers, we can insert an additional element and keep the array sorted in $O(n)$ time. The idea of insertion sort is to start with a sorted list consisting of the first element of some un-sorted list L. We then take the second element of L and insert it into the sorted list, which now has size two. Then we take the third element of L and insert it into the sorted list, which now has size three. We carry on this procedure for all elements of L. Algorithm 2 describes the full algorithm.

```
Data: list of elements L
Result: sorted list S of the elements in L
n ← number of elements in L
for i = 0 to n − 1 do
    j ← i
    while j > 0 and L(j − 1) > L(j) do
        swap L(j − 1) and L(j)
        j ← j − 1
    end
end
S ← L
```

Algorithm 2: Insertion sort sorting algorithm, `insertionsort(L)`

Note that if the array is already sorted, $L(j-1) > L(j)$ never holds and no swaps are done. Thus, the best case running time is $\Theta(n)$. If the elements are in reverse-sorted order, then all possible swaps are made. Thus, the running time is $1 + 2 + \ldots n - 1 = \Theta(n^2)$. The expected running time is also $\Theta(n^2)$ [CSRL01].

3.2 Merge sort

Merge sort makes use of an auxiliary function, `merge`, that takes two sorted lists, S_1 and S_2 of sizes n_1 and n_2 and creates a sorted list S of size $n_1 + n_2$. The function can be implemented in $O(n_1 + n_2)$ time. Merge sort is presented in Algorithm 3 and visualized in Figure 1.

Let $T(n)$ be the running time of merge sort on a list with n element. Since the merge routine takes $O(n)$ time,

$$T(n) = 2T(n/2) + O(n).$$

By Theorem 2.6, $T(n) = O(n \log n)$. We can see from Figure 1 that we always have $\Theta(\log n)$ steps of recursion. Assuming that the merge function looks at all of the elements (if, for example, the data are copied), then $\Theta(n)$ operations are used at each level of recursion. Thus, the expected, best-case, and worst-case running times are all $\Theta(n \log n)$.

Data: list of elements L
Result: sorted list S of the elements in L
$n \leftarrow$ number of elements in L
if $n = 1$ **then**
  $S \leftarrow L$
end
else
  $L_1 = [L(1), \ldots, L(n/2)]$
  $L_2 = [L(n/2+1), \ldots, L(n)]$
  $S_1 \leftarrow$`mergesort(`L_1`)`
  $S_2 \leftarrow$`mergesort(`L_2`)`
  $S \leftarrow$`merge(`S_1`, `S_2`)`
end

Algorithm 3: Merge sort sorting algorithm, `mergesort(L)`

3.3 Quicksort

Quicksort is similar in flavor to the k-select algorithm (Algorithm 1). We again rely on pivot elements, and the analysis is similar. Algorithm 4 describes quicksort.
In the worst case, the pivot could always be chosen as the smallest element, and $S_>$ would have size $n-1$, $n-2$, $\ldots$, 1 in some path of the recursion tree. This results in a worst-case running time of $\Theta(n^2)$. To get intuition for the average-case running time, we can follow the same analysis as in the analysis of k-select. In expectation, it takes two steps of recursion for the largest list to have size $3n/4$. Thus, the total number of steps in the recursive tree is $O(\log_{4/3} n) = O(\log n)$. Since $O(n)$ work is done in total at each step of the recursion tree, the expected running time is $O(n \log n)$. In the best case, the pivot always splits the data in half. Hence, the best case running time of $\Theta(n \log n)$.

Data: list of elements L
Result: sorted list S of the elements in L
pivot $\leftarrow$ uniform random element of L
$L_< \leftarrow \{x \in L | x < pivot\}$
$L_> \leftarrow \{x \in L | x > pivot\}$
$L_= \leftarrow \{x \in L | x = pivot\}$
$S_< \leftarrow$ `quicksort(`$L_<$`)`
$S_> \leftarrow$ `quicksort(`$L_>$`)`
$S \leftarrow [S_<, S_=, S_>]$

Algorithm 4: Quicksort algorithm

We now introduce the notion of stability in sorting algorithms.

Definition 3.1. Consider a sorting algorithm that sorts a list of elements of L and outputs a sorted list S of the same elements. Let $f(i)$ be the index of the element $L(i)$ in the sorted list S. The sorting algorithm is said to be *stable* if $L(i) = L(j)$ and $i < j$ imply that $f(i) < f(j)$.

Note that insertion sort and merge sort are stable as described. Typical efficient implementations of quick sort are not stable.

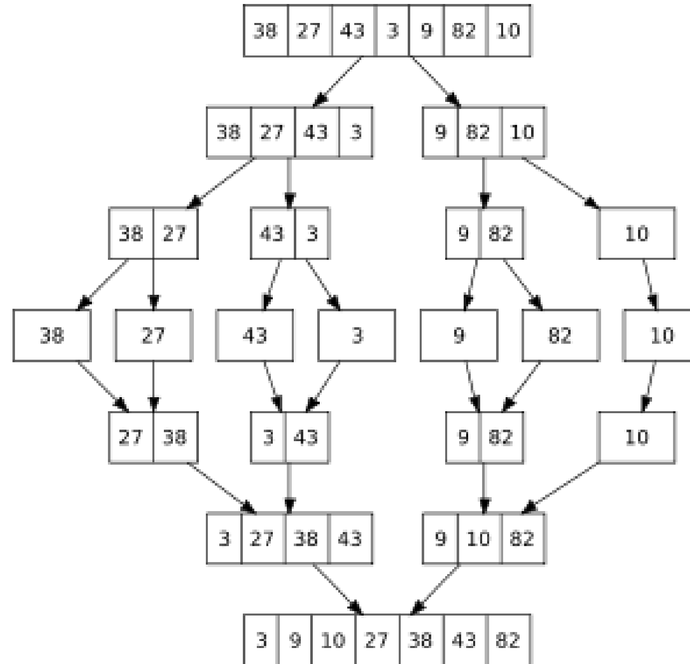

Figure 1: Illustration of the recursion in merge sort. Image from http://i.stack.imgur.com/4tsCo.png.

Table 1 summarizes the sorting algorithms we have covered so far in this section. There are a host of other sorting algorithms, and we have only brushed the surface of the analysis. How do we choose which algorithm to use in practice? Often, we don't need to choose an efficient implementations are already provided, e.g., the C++ standard template library functions `std::sort` and `std::stable_sort`. Quicksort is typically fast in practice and is a common general-purpose sorting algorithm. Also, quicksort has a small memory footprint, which is something we have not considered. However, quicksort is not stable, which might be desirable in practice. Insertion sort is nice for when the array is nearly sorted.

Table 1: Summary of sorting algorithms.

Algorithm	worst-case	best-case	expected	stable?
Insertion sort	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	yes
Merge sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	yes
Quicksort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	no

3.4 Bucket sort and radix sort

We have so far been bounded by $\Theta(n \log n)$ complexity. The reason is that we have only considered *comparison-based sorting algorithms*, i.e., algorithms that depend only on a comparison function that determines whether one element is larger than another.
Suppose have a list L of numbers in $\{0, 1, \ldots, 9\}$. We want to sort these numbers in $\Theta(n)$ time.

We can create ten buckets—one for each digit. For each element in the list, we simply add it to the corresponding bucket. The sorted list is then the concatenation of the buckets. This process is described in Algorithm 5. Using, for example, a linked list to implement the buckets, bucket sort is stable. The running time of bucket sort is $\Theta(n)$, but the catch is that the elements all had to be digits. Note that bucket sort makes no comparisons!

Data: list of numbers L with each number in $\{0, 1, \ldots, 9\}$
Result: sorted list S of the numbers in L
$n \leftarrow$ number of elements in L
Create buckets $B_0, \ldots, B_9$. **for** $i = 1$ *to* n **do**
  Append $L(i)$ to bucket $B_{L(i)}$.
end
$S \leftarrow [B_0, \ldots, B_9]$

Algorithm 5: Bucket sort algorithm

We will now leverage bucket sort to implement radix sort. Suppose that we have a list of L numbers in $\{0, 1, \ldots, 10^d - 1\}$. We can use Radix sort (Algorithm 6) to sort this list efficiently. The cost of each iteration of the j index takes $\Theta(n)$ time. Thus, the total running time is $\Theta(dn)$. Note that since the bucket sort is stable, radix sort is also stable.

Data: list of numbers L with each number in $\{0, 1, \ldots, 10^d - 1\}$
Result: sorted list S of the numbers in L
$n \leftarrow$ number of elements in L
for $j = 1$ *to* d **do**
  `// 1 is least significant digit, d is most significant`
  $L \leftarrow$`bucketsort(L)` using j-th digit to place in buckets
end
$S \leftarrow L$

Algorithm 6: Radix sort algorithm

Radix sort can be used to handle more general sorting routines. In general, we can consider the elements of L to be length-d tuples such that each element of the tuple has at most k unique values. Then the running time of radix sort is $\Theta(d(n + k))$.

4 Basic graph theory and algorithms

References: [DPV06, Ros11].

4.1 Basic graph definitions

Definition 4.1. A *graph* $G = (V, E)$ is a set V of *vertices* and a set E of *edges*. Each edge $e \in E$ is associated with two vertices u and v from V, and we write $e = (u, v)$. We say that u *is adjacent to* v, u *is incident to* v, and u *is a neighbor of* v.

Graphs are a common abstraction to represent data. Some examples include: road networks, where the vertices are cities and there is an edge between any two cities that share a highway; protein

interaction networks, where the vertices are proteins and the edges represent interactions between proteins; and social networks, where the nodes are people and the edges represent friends.
Sometimes we want to associate associate a direction with the edges to indicate a one-way relationship. For example, consider a predator-prey network where the vertices are animals and an edge represents that one animal hunts the other. We want to express that the fox hunts the mouse but not the other way around. This naturally leads to *directed graphs*:

Definition 4.2. A *directed graph* $G = (V, E)$ is a set V of *vertices* and set E of *edges*. Each edge $e \in E$ is an ordered pair of vertices from V. We denote the edge from u to v by (u, v) and say that *u points to v*.

Here are some other common definitions that you should know:

Definition 4.3. A *weighted graph* $G = (V, E, w)$ is a graph (V, E) with an associated weight function $w : E \to \mathbb{R}$. In other words, each edge e has an associated weight $w(e)$.

Definition 4.4. In an undirected graph, the *degree* of a node $u \in V$ is the number of edges $e = (u, v) \in E$. We will write d_u to denote the degree of node u.

Lemma 4.5. ***The handshaking lemma***. *In an undirected graph, there are an even number of nodes with odd degree.*

Proof. Let d_v be the degree of node v.

$$2|E| = \sum_{v \in G} d_v = \sum_{v \in G : d_v \text{ even}} d_v + \sum_{v \in G : d_v \text{ odd}} d_v$$

The terms $2|E|$ and $\sum_{v \in G : d_v \text{ even}} d_v$ are even, so $\sum_{v \in G : d_v \text{ odd}} d_v$ must also be even. □

Definition 4.6. In a directed graph, the *in-degree* of a node $u \in V$ is the number of edges that point to u, i.e., the number of edges $(v, u) \in E$. The *out-degree* of u is the number of edges that point away from u, i.e., the number of edges $(u, v) \in E$.

Definition 4.7. An edge (u, u) in a graph is called a *self-edge* or *loop*.

Definition 4.8. A *path* on a graph is a sequence of nodes $v_1, v_2, \ldots v_k$ such that the edge $(v_i, v_{i+1}) \in E$ for $i = 1, \ldots, k - 1$.

Definition 4.9. A *cycle* is a path $v_1, v_2, \ldots v_k$ with $k \geq 3$ such that $v_1 = v_k$.

4.2 Storing a graph

How do we actually store a graph on a computer? There are two common methods: an *adjacency matrix* or an *adjacency list*. In the adjacency matrix, we store a matrix A of size $|V| \times |V|$ with entry $A_{ij} = 1$ if edge (i, j) is in the graph, and $A_{ij} = 0$ otherwise. The main advantage of an adjacency matrix is that we can query the existence of an edge in $\Theta(1)$ time—we just look up the entry in the matrix. The disadvantage is that storage is $\Theta(|V|^2)$. In many graphs, the number of edges is far less than $|V|^2$, which makes adjacency matrices impractical for many problems. However, adjacency matrices are still a useful tools for understanding graphs.
Adjacency lists, on the other hand, only need memory proportional to the size of the graph. They consist of a length-$|V|$ array of linked lists. The ith linked list contains all nodes adjacent to node i (also called node i's *neighbor list*). We can access the neighbor list of i in $\Theta(1)$ time, but querying for an edge takes $O(d_{\max})$ time, where $d_{\max} = \max_j d_j$.

4.3 Graph exploration

Definition 4.10. We say that node v is *reachable* from node u if there is a path from u to v.

We now cover two ways of exploring a graph: depth-first search (DFS) and breadth-first search (BFS). The goal of these algorithms is to find all nodes reachable from a given node. We will describe the algorithms for undirected graphs, but they generalize to directed graphs.

4.3.1 Depth-first search

This construction of DFS comes from [DPV06]. We will use an auxiliary routine called `Explore(v)`, where v specifies a starting node. We use a vector C to keep track of which nodes have already been explored. The routine is described in Algorithm 7. We have included functions `Pre-visit()` and `Post-visit()` before and after exploring a node. We will reserve these functions for book-keeping to help us understand other algorithms. For example, `Pre-visit()` may record the order in which nodes were explored.

```
Data: Graph G = (V, E), starting node v, vector of covered nodes C
Result: C[v] = true for all nodes reachable from v
C[v] ← true
for (v, w) ∈ E do
    if C[w] is not true then
        Pre-visit(w)
        Explore(w)
        Post-visit(w)
    end
end
```

Algorithm 7: Function `Explore(v)`

Given the explore routine, we have a very simple DFS algorithm. We present it in Algorithm 8. The cost of DFS is $O(|V| + |E|)$. We only call `Explore()` on a given node once. Thus, we only consider the neighbors of each node once. Therefore, we only look at the edges of a given node once.

```
Data: Graph G = (V, E)
Result: depth-first-search exploration of G
for v ∈ V do
    // Initialized covered nodes
    C[v] ← false
end
for v ∈ V do
    if C[v] is not true then
        Explore(v)
    end
end
```

Algorithm 8: Function `DFS(G)`

We now look at an application of DFS: topological ordering of DAGs.

Definition 4.11. A *directed acyclic graph (DAG)* is a directed graph that has no cycles.

DAGs characterize an important class of graphs. One example of a DAG is a schedule. There is an edge (u, v) if task u must finish before task v. If a cycle exists, then the schedule cannot be completed! We now investigate how to order the tasks to create a valid sequential schedule.

Definition 4.12. A *topological ordering* of the nodes v is an order such that if `order[v]` $<$ `order[w]`, then there is no path from w to v.

Example 4.13. *Suppose we define the post-visit function to simply update a count and record the counter, as in Algorithm 9. Ordering the nodes by decreasing* `post[v]` *after a DFS search gives a topological ordering of a DAG.*

Proof. Suppose not. Then there are nodes v and w such that post$[v] >$ post$[w]$ and there is a path from w to v. If we called the explore function on w first, then post$[v] <$ post$[w]$ as there is a path from w to v. This is a contradiction. Suppose instead that we called the explore function on v first. If w was reachable from v, then there is a cycle in the graph. Hence, w must not be reachable from v. But then post$[w]$ would be larger than post$[v]$ since we explored v first. Again, we have a contradiction. □

```
// Post-visit()
// count is initialized to 0 before running the algorithm
post[v] ← count
count ← count + 1
```

Algorithm 9: Post-visit function to find topological orderings

4.3.2 Breadth-first search

With DFS, we went as far along a path away from the starting node as we could. With breadth-first-search (BFS), we instead look at all neighbors first, then neighbors of neighbors, and so on. For BFS, we use a queue data structure (see Appendix of previous year's notes, on website). A *queue* is an object that supports the following two operations:

- `enqueue(x)`: puts the data element x at the back of the queue
- `dequeue()`: returns the oldest element in the queue

Algorithm 10 describes BFS and also keeps track of the distances of nodes from a source node s to all nodes reachable from s.

```
Data: Graph G = (V, E), source node s
Result: For all nodes t reachable from s, dist[t] is set to the length of the smallest path
        from s to t. dist[t] is set to ∞ for nodes not reachable from s
for v ∈ V do
 | dist[v] ← ∞
end
dist[s] ← 0
Initialize queue Q
Q.enqueue(s)
while Q is not empty do
 | u ← Q.dequeue()
 | for (u, v) ∈ E do
 | | if dist[v] is ∞ then
 | | | Q.enqueue(v)
 | | | dist[v] ← dist[u] + 1
 | | end
 | end
end
```

Algorithm 10: Breadth-first search

4.4 Connected components

Definition 4.14. An *induced subgraph* of $G = (V, E)$ specified by $V' \subset V$ is the graph $G' = (V', E')$, where $E' = \{(u, v) \in E | u, v \in V'\}$.

Definition 4.15. In an undirected graph G, a *connected component* is a maximal induced subgraph $G' = (V', E')$ such that for all nodes $u, v \in V'$, there is a path from u to v in G'.

By "maximal", we mean that there is no V'' such that $V' \subset V''$ and the induced subgraph specified by V'' is also a connected component.

Definition 4.16. In a directed graph G, a *strongly connected component* is a maximal induced subgraph $G' = (V', E')$ such that for all nodes $u, v \in V'$, there are path from u to v and from v to u in G'.

Definition 4.17. An undirected graph is *connected* if G is a connected component. A directed graph is *strongly connected* if G is a strongly connected component.

We now present a simple algorithm for finding connected components in an undirected graph. First, we update the DFS procedure (Algorithm 8) to increment a counter `count` before calling `Explore()`, with initialization to 0. Then the previsit function assigns connected component numbers.

```
// Pre-visit()
connected_component[v] ← count
```

Algorithm 11: Pre-visit for connected components in undirected graphs

4.5 Trees

Definition 4.18. A *tree* is a connected, undirected graph with no cycles.

Definition 4.19. A *rooted tree* is a tree in which one node is called the root, and edges are directed away from the root.

Note that all trees can be rooted tree from any vertex in the tree since there are no cycles.

Definition 4.20. In a rooted tree, for each directed edge (u, v), u is the *parent* of v and v is the *child* of u.

Note that the parent of a node v must be unique.

Definition 4.21. In a rooted tree, a *leaf* is a node with no children.

Note that all finite trees must have at least one leaf node (we can find them by a topological ordering). Can also see this fact by starting at an arbitrary vertex and following any path, which must stop somewhere, and where it does, we have found a leaf node. In fact, every tree has at least 2 leaf nodes, because we can start from the leaf node instead of an arbitrary node as before; following an arbitrary path necessarily leads to a second leaf node (cannot be the same as the original because no cycles).

Theorem 4.22. *A tree with n vertices has $n - 1$ edges.*

Proof. We go by induction. With $n = 1$, there are no edges. Consider a graph with k nodes. Let v be a leaf node and consider removing v and (u, v) from the tree, where u is the parent of v. The remaining graph is a tree, with one fewer edge. By induction, it must have $k - 2$ edges. Therefore, the original tree had $k - 1$ edges. □

4.5.1 Binary trees

Definition 4.23. A *binary tree* is a tree where every node has at most two children.

In a binary tree, there are a few natural ways to order the nodes (or *traverse* the tree). We describe them in Algorithm 12.

Exercise 4.24. *Which traversals in Algorithm 12 correspond to DFS and BFS?*

4.5.2 Minimum spanning trees

Definition 4.25. A spanning tree of an undirected graph $G = (V, E)$ is a tree $T = (V, E')$, where $E' \subset E$.

Definition 4.26. The minimum spanning tree $T = (V, E')$ of a weighted, connected undirected graph $G = (V, E)$ is a spanning tree where the sum of the edge weights of E' is minimal.

Two algorithms for finding minimum spanning trees are Prim's algorithm (Algorithm 13) and Kruskal's algorithm (Algorithm 14). Both procedures are straightforward. We will analyze Kruskal's algorithm for correctness to get used to this type of analysis. Note that you will go over Kruskal's algorithm in CME 305 and by the end of the class you will see more generally why it works (because greedy algorithms work for matroid optimazation).

Data: Node v
Result: Traversal of subtree rooted at node v
$v_l \leftarrow$ left child (if it exists) $v_r \leftarrow$ right child (if it exists)
`// Post-order`
Post-order(v_l)
Post-order(v_r)
Visit(v)

`// Pre-order`
Visit(v)
Pre-order(v_l)
Pre-order(v_r)

`// In-order`
In-order(v_l)
Visit(v)
In-order(v_r)

Algorithm 12: Tree traversal algorithms

Data: Connected weighted undirected graph $G = (V, E, w)$
Result: Minimum spanning tree $T = (V, E')$
$S \leftarrow \{v\}$ for any arbitrary node $v \in V$
$E' \leftarrow \emptyset$
while $|S| < |V|$ **do**
  $(u, v) \leftarrow \arg\min_{u \in S, v \notin S} w((u, v))$
  $S \leftarrow S \cup \{u\}$
  $E \leftarrow E' \cup \{(u, v)\}$
end

Algorithm 13: Prim's algorithm for minimum spanning trees

Data: Connected weighted undirected graph $G = (V, E, w)$
Result: Minimum spanning tree $T = (V, E')$
$E' \leftarrow \emptyset$
foreach $e = (u, v) \in E$ *by increasing* $w(e)$ **do**
  if *adding* (u, v) *to* E *does not creates a cycle* **then**
    $E' \leftarrow E' \cup \{(u, v)\}$
  end
end

Algorithm 14: Kruskal's algorithm for minimum spanning trees

Theorem 4.27. *The output, T, from Kruskal's algorithm is a minumum weight spanning tree for the connected graph G.*

Proof. First we notice that T must be a spanning tree of G because by construction it has no cycles and must be connected and spanning. Otherwise we would have multiple connected components, and because G is connected the algorithm must have considered at least one edge connecting these disconnected components and added at least one (no cycles would have formed). So T is a spanning tree of G.
Now we must show that it is a minumum weight spanning tree. Suppose not, so there exists a minimum spanning tree T' of G such that $w(T') < w(T)$. There must exist some edge $e \in T'$ such that $e \notin T$. Consider $T \cup \{e\}$, which must contain a cycle now because it has n edges. Call this cycle C. Because $e \notin T$, when we considered e, we must have already added all other edges in C to T (otherwise e would be in T). Therefore, $w(f) \leq w(e)\ \forall f \neq e \in C$. Note that we are assuming here that all edge weights are distinct; if they are not, the proof is very similar.
But $T'' = T' \cup \{f\} \setminus \{e\}$ is also a spanning tree of G such that $w(T'') < w(T')$, so T' cannot be a minumum spanning tree of G with weight less than T. □

Note that if we use the correct data structures for our graph, the major step in this algorithm is sorting the edges by weight. Therefore, the running time of Kruskal's algorithm is $O(|E|\log(|E|)$.

4.6 Cycles

Definition 4.28. A *Hamiltonian cycle* of a graph G is a cycle that visits each node of G exactly once.

Definition 4.29. A *Hamiltonian path* of a graph G is a path that visits each node of G exactly once.

Definition 4.30. A *Eulerian cycle* of a graph G is a cycle that visits each edge of G exactly once.

Definition 4.31. A *Eulerian path* of a graph G is a path that visits each edge of G exactly once.

Exercise 4.32. *Show the following:*

- *A graph is Eulerian (has an Eulerian cycle) if and only if the degree of every vertex is even.*
- *A graph contains an Eulerian path if and only if it has exactly two vertices of odd degree.*

Example 4.33. *Suppose the minimum degree of a graph G is k. Show that the graph contains a cycle of length at least $k + 1$.*

Proof. Consider the longest path in G. Let $u, v \in V$ be the endpoints of this path. We know that all the neighbors of u and v must be on this path, otherwise this would not be the longest path. So necessarily, both u and v must be in cycles of length at most $k + 1$ (easy to see by drawing a picture). □

Note that this technique of considering the longest path of a graph when working under assumptions about the minimum degree can be very useful. You will see in class that using this technique (along with some other ideas) that if the minimum degree of a graph is greater than or equal to $\frac{|V|}{2}$, then the graph contains a Hamiltonian cycle, usually a very hard property to determine.

Exercise 4.34. *If C is a cycle, and e is an edge connecting two nonadjacent nodes of C, then we call e a* chord. *Show that if every node of a graph G has degree at least 3, then G contains a cycle with a chord.*

4.7 Cuts

Definition 4.35. A *cut* $C = (A, B)$ of a graph $G = (V, E)$ is a partition of V into two subsets A and B. The *cut-set* of C is the set of edges that cross the cut, i.e. $\{(u, v) \in E | u \in A, v \in B\}$.

We are almost always interested in the number of edges (or total weight of the edges in the case of directed graphs) in the cut-set.

Definition 4.36. An *s-t minimum cut* is the cut, $C^* = (S, T)$, of minimum weight that separates two vertices s and t so that $s \in S$ and $t \in T$.

Definition 4.37. A *global minimum cut* is a cut of minimum weight, $C^* = (A, B)$, in the graph G such that both A and B are non-empty.

Definition 4.38. A *global maximum cut* is a cut of maximum weight, $C^* = (A, B)$, in the graph G such that both A and B are non-empty.

You will see in class that $s - t$-minimum cuts are very related to network flow problems and are very useful for scheduling problems among other applications. Both kinds of minumum cuts are easy to compute, while maximum cuts are very hard.

4.8 Other definitions and examples

Definition 4.39. A *bipartite graph* is a graph whose vertices can be divided into two disjoint sets A and B such that every edge connects a vertex in A to one in B.

Although we said earlier that max cuts are typically difficult to compute, they are very easy for bipartite graphs.

Exercise 4.40. *Show that every tree is bipartite.*

Note that we can further prove that a graph is bipartite if and only if it contains no cycles of odd length.

Definition 4.41. An *independent set* is a set of vertices $I \subseteq V$ in a graph such that no two in the set are adjacent.

Definition 4.42. A *vertex cover* of a graph is a set of vertices $C \subseteq V$ such that each edge of the graph is incident to at least one vertex in C.

Definition 4.43. An *edge cover* of a graph is a set of edges $C \subseteq E$ such that every vertex of the graph is incident to at least one edge in C.

Note that we can see immediately from the above definitions that each side of a bipartite graph will necessarily be both an independent set and a vertex cover. Using this fact and the above exercise, we can see also that every tree on n nodes has a vertex cover of size at most $\lceil \frac{n-1}{2} \rceil$.

Definition 4.44. A *matching* M in a graph G is a set of edges such that no two edges share a common vertex.

Definition 4.45. A *maximal matching* is a matching M of a graph G with the property that if any edge of G not already in M is added to M, the set is no longer a valid matching.

Definition 4.46. A *maximum matching* of a graph G is the matching that contains the most number of edges.

We see that all maximum matchings must also be maximal, but the opposite is certainly not true.

Definition 4.47. A *perfect matching* of a graph G is a matching such that all vertices of G are incident to an edge in the matching.

Example 4.48. *Assume that you are given a graph $G = (V, E)$, a matching M in G and an independent set S in G. Show that $|M| + |S| \leq |V|$.*

Proof. Let M^* be a maximum matching of G and S^* be a maximum independent set of G. For each of the $|M^*|$ matched pairs, $|S^*|$ can include at most one of the vertices, so $|S^*| \leq |V| - |M^*|$. We have $|S^*| + |M^*| \leq |V|$ and because M^* and S^* are the maximum size matching and independent set respecitively, we have $|S| + |M| \leq |V|$ holds for all matchings and independent sets. $\square$

5 Counting

References: [Ros11]

Definition 5.1. Let $k \leq n$ be integers. An *ordered arrangement* of k elements from a set of n distinct elements is called a *k-permutation*. An *un-ordered collection* of k elements from a set of n distinct elements is called an *k-combination*.

Theorem 5.2. *The number of k-permutations from a set of n elements is $\frac{n!}{(n-k)!}$, and the number of k-combinations is $\frac{n!}{k!(n-k)!}$.*

We write $\frac{n!}{k!(n-k)!}$ as $\binom{n}{k}$ and say it as "n choose k". As a sanity check, note that if $k = 1$, both terms simplify to n, corresponding to picking one of the elements from the set. And if $k = n$, the number of k-combinations is one—you have to select all elements for the combination. Also, we see that the number of k-permutations is $k!$ multiplied by the number of k-combinations. This corresponds to the $k!$ ways of ordering the elements in the un-ordered combination. Now, an informal proof of Theorem 5.2:

Proof. There are n ways to pick the first element, $n - 1$ to pick the second element, and so on. Thus, there are $n \cdot (n-1) \cdot \ldots \cdot (n-k+1) = \frac{n!}{(n-k)!}$ k-permutations. There are $k!$ ways to order k elements, so there are $\frac{n!}{k!(n-k)!} = \binom{n}{k}$ k-combinations. $\square$

Theorem 5.3. ***The binomial theorem.***

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$$

Proof. All terms in the product are of the form $x^{n-k}y^k$ for some k. For a fixed k, we count how many ways we get a term $x^{n-k}y^k$. We have a collection of n instances of the term $(x + y)$, and choosing exactly k "y terms" gives us y^k. The number of ways to choose exactly k "y terms" is a k-combination from a set of n elements. □

The $\binom{n}{k}$ term is called the *binomial coefficient*. The binomial theorem is a clever way of proving interesting polynomial and combinatorial equalities. For example:

Example 5.4.

$$\sum_{k=0}^{n} \binom{n}{k} (-1)^k = 0$$

To see this, apply the binomial theorem with $x = 1$ and $y = -1$.

We end this section with an "obvious" principle, but it is worth remembering, which we will first show with a quick example.

Example 5.5. *Every simple graph on n vertices with $n \geq 2$ has two vertices of the same degree.*

Proof. We go case by case by the number of zero-degree nodes. If G has no zero-degree nodes, then we have n nodes all with degree between 1 and $n - 1$, so at least two of these nodes must have the same degree. If G has a single zero-degree node, then the other $n - 1$ nodes must have degree between 1 and $n - 2$ so again at least two must have the same degree. If G has more than one zero-degree node, we are done. □

The above proof uses the pigeonhole principle.

Theorem 5.6. ***The pigeonhole principle***. *If $n \geq k + 1$ objects are put into k containers, then there is a container with at least two objects.*

You can prove the pigeonhole principle by contradiction. An extension of the principle is:

Theorem 5.7. ***The generalized pigeonhole principle***. *If n objects are put into k containers, then there is a container with at least $\lceil n/k \rceil$ objects.*

6 Randomized Algorithms

A randomized algorithm is an algorithm that uses a degree of randomness as part of its logic. We often use randomized algorithms in hopes of achieving good results in expectation. Note that we have already seen a couple examples of randomized algorithms, the k-select algorithm and quicksort, which both performed well in expectation. Randomized algorithms are very useful when input order can dramatically affect the performance of a deterministic algorithm (e.g. an adversary can provide a bad input to a deterministic very easily), so simply randomizing any input can help performance in expectation.

There are two main types of randomized algorithms.

Definition 6.1. A *Las Vegas* algorithm is a randomized algorithm that always outputs a correct result but the time in which it does so is a random variable.

Definition 6.2. A *Monte Carlo* algorithm is a randomized algorithm with deterministic run-time but some probability of outputting the incorrect result.

What kind of algorithm is quicksort? Note that we can turn any Las Vegas algorithm into a Monte Carlo algorithm by simply imposing a maximum run-time and outputting the current guess. Now we have a deterministic run-time and some probability of error.

Often we may be using a randomized algorithm to answer a decision problem, such as whether or not two polynomials are equal. We may distinguish between a Monte Carlo algorithm with one-sided error and two-sided error. For example, an algorithm with one-sided error may always output a correct result for a certain type of input (yes or no for the decision problem), but then have some probability of error for the other type of input, while an algorithm with two-sided error will have some probability of error for every input.

Exercise 6.3. *Suppose we have three $n \times n$ matrices A, B and C. Develop an algorithm that determines if $C = AB$ in $O(n^2)$ time with high probability.*

Notice that the algorithm to solve the above exercise is a one-sided Monte Carlo algorithm (it is called Freivalds algorithm). We can always repeat our algorithm a constant number times to boost the probability of success but keep the asymptotic run time the same.

6.1 Probabilistic Method

One topic related to randomized algorithms is the probabilistic method, a very powerful non-constructive proof technique developed by Paul Erdos to show existence of an object/instance. To show the existence of some property or object, we only need to create an appropriate probability space over all possible objects and show that our specific object occurs with a non-zero probability.

Exercise 6.4. *Let $G = (V, E)$ be a bipartite graph on n vertices with a list $S(v)$ of at least $1 + \log_2 n$ colors associated with each vertex $v \in V$. A graph coloring is an assignment of colors to nodes such that no two adjacent nodes have the same color. Prove that there is a proper coloring of G assigning to each vertex of v a color from its list $S(v)$.*

7 Complexity Theory

The field of complexity theory deals with classifying computational problems according to their inherent difficulty and understanding how these classes of problems relate with each other. Knowing the given complexity of a problem (or being able to show it) can help us understand how to solve a problem. If we are having trouble developing an efficient algorithm for a particular problem, we may find it useful to prove the complexity of the problem; if it is a very difficult problem, we know we cannot hope to solve correctly in an efficient manner so we should be looking for other options.

(Taken from course notes for CME 305) We will be talking about decision problems. We can represent a particular instance of a decision problem as an input string s, which encodes all the information about the instance of the problem we are interested in. Let X be the set of all problem instances represented by strings s that are solvable. A decision problem asks if $s \in X$.

We define P as the class of decision problems that can be correctly decided in polynoial time, i.e. there exists some deterministic verifier $A(s)$ that will return T iff $s \in X$ in polynomial time.

We define NP as the class of decision problems for which a *yes* certificate can be verifies in polynomial time. That is, if there exists a polynomial-time deterministic verifier $B(s,t)$ such that if $s \in X$, there is some $t : |t| \leq |s|$ (called a *certificate*) such that $B(s,t)$ returns T, otherwise (if $s \notin X$) $B(s,t)$ returns F for all t.

Any problem X is NP-hard if for all problems $Y \in NP$, $Y \leq_p X$. That is Y is poly-time reducible to X, so if we have a black-box algorithm to solve problem X, we may solve every instance of Y with a polynomial amount of work and polynomial number of calls to the black box.

A problem is NP-complete if it is in NP and is NP-hard.

Stephen Cook first showed the existence of NP-complete problems, by showing the circuit satisfiability problem (SAT) is NP-complete. We will not go into the details of this problem or its proof at this point. We may use this result now though to show that other problems are NP-complete as well, all need to do is reduce a problem to the circuit satisfiability problem, showing that it is at least as hard as this it and therefore must also be NP-hard.

We now have a great list of problems that are NP-hard, which include SAT, $3 - SAT$, the vertex cover decision problem, the independent set decision problem, integer programming and many more. Do not worry about the details of these problems right now, you will see them in much greater detail in the future.

The vertex cover decision problem asks whether a graph has a vertex cover of size at most k where G and k are the input to the problem. The independent set decision problem asks whether a graph has an independent set of size at least k where again G and k are the input to the problem.

Exercise 7.1. *Show that the vertex cover problem and independent set problem are equivalent, that is we can reduce either one to the other.*

8 Approximation Algorithms

How do we solve problems that we know are in NP then? Well, as we saw previously, some problems are not hard on all inputs (e.g. max cut for bipartite graphs, the Hamiltonian cycle problem on graphs with certain min degree properties). Additionally, for some problems we may develop algorithms based on various heuristics that may not work well on every input but work well enough for some subset of the input, particularly ones in which we are most interested. The most rigorous way to deal with these hard problems though is a field called approximation algorithms.

Approximation algorithms are algorithms that find approximate solutions (we usually care that they do this in polynomial time) to optimization problems, most often hard problems, such that the result is within some proven factor of the optimal solution. Let OPT be the optimal solution to some problem and O be the output of some approximation algorithm to that problem.We say that

the algorithm is a k-approximation if $OPT \leq O \leq k \cdot OPT$ for $k > 1$ (a minimization problem) or $k \cdot OPT \leq O \leq OPT$ for $k < 1$ (a maximation problem). We will go through a couple examples.

Example 8.1. *Develop a simple* $1/2$*-approximation (in expectation) for computing the max-cut of an undirected graph* G*.*

In this case, we only care about getting the approximation factor in expectation. Consider the following procedure:
For each vertex, flip a fair coin.
If heads, put v in A. Else, put v in B.

Note that we can also put in a line to redo the random assignment if one of the partition sets is empty, but this should never be a problem. Now we must show that in expectation that this procedure will give a 1/2-approximation.

Let $C = (A, B)$ be the cut output from the procedure. Let X_e be the indicator variable that edge $e \in E$ is across the cut after partitioning the vertex-set. We know that $e = (u, v)$ will be across the cut if and only if u is in one side of the partition and v is in the other. This results happens with a probability 0.5. So we have:

$$\mathrm{E}[w(C)] = \sum_{e \in E} \mathrm{E}[X_e] \cdot w(e) = \frac{1}{2} \sum_{e \in E} w(e) \geq \frac{1}{2} \cdot OPT$$

We have used the fact that the optimal maximum cut must be less than or equal to the total sum of the edge weights.

Consider the problem of finding a minimum size vertex cover VC, which we know to be NP-hard. We may actually write VC as an integer program. Create an indicator variable X_v for every $v \in V$. We say that if $X_v = 1$, v will be in our vertex cover and if $X_v = 0$ it is not. Then our integer program for VC problem is simply:

$$\begin{aligned} \underset{X_v}{\text{minimize}} \quad & \sum_{v \in V} X_v \\ \text{subject to} \quad & X_v \in \{0, 1\} \ \forall v \in V, \\ & X_u + X_v \geq 1 \ \forall e = (u, v) \in E. \end{aligned}$$

The above integer program exactly solves the minimum vertex cover problem, and it is again NP-hard to solve (integer programming is NP-hard). However, a very common method used in approximation algorithms is to relax integer programs into linear programs or other convex programs that are efficient to solve. In this case, all we need to do is relax the constraint on X_v from an integer constraint to a linear one, so instead we will have the above optimization problem with $0 \leq X_v \leq 1 \ \forall v \in V$ as a constraint.

Exercise 8.2. *Show that the above linear program relaxation gives a 2-approximation algorithm to solve* VC*.*

In this case you will end up using the fact that the minimum of the LP cannot ever be greater than the mimum of the IP because the feasible region of the LP strictly contains that of the IP.

Notice that in both of these examples we have needed a handle on OPT. We do not have a good idea of OPT for the problem (otherwise the problem would be easy to solve), so we need a good way to bound the value to help our analysis. In the first case, we saw that OPT can never be more than the total weight of the edges. In the second, we use the fact that relaxing a IP increases the size of the feasible region allowing us to use the optimum of our new LP to bound the true OPT. Choosing how to handle OPT to give a clean analysis is very important when developing approximation algorithms.

References

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd edition, 2001.

[DPV06] Sanjoy Dasgupta, Christos H Papadimitriou, and Umesh Vazirani. *Algorithms.* McGraw-Hill, Inc., 2006.

[Ros11] Kenneth Rosen. *Discrete Mathematics and Its Applications 7th edition.* McGraw-Hill Science, 2011.