Lecture 1 Algorithms and Their Complexity

This is a course on the design and analysis of algorithms intended for first-year graduate students in computer science. Its purposes are mixed: on the one hand, we wish to cover some fairly advanced topics in order to provide a glimpse of current research for the benefit of those who might wish to specialize in this area; on the other, we wish to introduce some core results and techniques which will undoubtedly prove useful to those planning to specialize in other areas.

We will assume that the student is familiar with the classical material normally taught in upper-level undergraduate courses in the design and analysis of algorithms. In particular, we will assume familiarity with:

- sequential machine models, including Turing machines and random access machines (RAMs)
- discrete mathematical structures, including graphs, trees, and dags, and their common representations (adjacency lists and matrices)
- fundamental data structures, including lists, stacks, queues, arrays, balanced trees
- fundamentals of asymptotic analysis, including $O(\cdot)$, $o(\cdot)$, and $\Omega(\cdot)$ notation, and techniques for the solution of recurrences
- fundamental programming techniques, such as recursion, divide-and-conquer, dynamic programming
- basic sorting and searching algorithms.

These notions are covered in the early chapters of [3, 39, 100].

Familiarity with elementary algebra, number theory, and discrete probability theory will be helpful. In particular, we will be making occasional use of the following concepts: linear independence, basis, determinant, eigenvalue, polynomial, prime, modulus, Euclidean algorithm, greatest common divisor, group, ring, field, random variable, expectation, conditional probability, conditional expectation. Some excellent classical references are [69, 49, 33].

The main emphasis will be on *asymptotic worst-case complexity*. This measures how the worst-case time or space complexity of a problem grows with the size of the input. We will also spend some time on probabilistic algorithms and analysis.

1.1 Asymptotic Complexity

Let f and g be functions $\mathcal{N} \to \mathcal{N}$, where $\mathcal{N}$ denotes the natural numbers $\{0, 1, \ldots\}$. Formally,

- f is $O(g)$ if

$$\exists c \in \mathcal{N} \quad \overset{\infty}{\forall} n \; f(n) \leq c \cdot g(n) \; .$$

  The notation $\overset{\infty}{\forall}$ means "for almost all" or "for all but finitely many". Intuitively, f grows no faster asymptotically than g to within a constant multiple.

- f is $o(g)$ if

$$\forall c \in \mathcal{N} \quad \overset{\infty}{\forall} n \; f(n) \leq \frac{1}{c} \cdot g(n) \; .$$

  This is a stronger statement. Intuitively, f grows strictly more slowly than any arbitrarily small positive constant multiple of g. For example, n^{347} is $o(2^{(\log n)^2})$.

- f is $\Omega(g)$ if g is $O(f)$. In other words, f is $\Omega(g)$ if

$$\exists c \in \mathcal{N} \quad \overset{\infty}{\forall} n \; f(n) \geq \frac{1}{c} \cdot g(n) \; .$$

- f is $\Theta(g)$ if f is both $O(g)$ and $\Omega(g)$.

There is one cardinal rule:

Always use O and o for upper bounds and Ω for lower bounds. *Never* use O for lower bounds.

There is some disagreement about the definition of Ω. Some authors (such as [43]) prefer the definition as given above. Others (such as [108]) prefer: f is $\Omega(g)$ if g is not $o(f)$; in other words, f is $\Omega(g)$ if

$$\exists c \in \mathcal{N} \quad \overset{\infty}{\exists}\, n \;\; f(n) > \frac{1}{c} \cdot g(n) \;.$$

(The notation $\overset{\infty}{\exists}$ means "there exist infinitely many".) The latter is weaker and presumably easier to establish, but the former gives sharper results. We won't get into the fray here, but just comment that neither definition precludes algorithms from taking less than the stated bound on certain inputs. For example, the assertion, "The running time of mergesort is $\Omega(n \log n)$" says that there is a c such that for all but finitely many n, there is some input sequence of length n on which mergesort makes at least $\frac{1}{c} n \log n$ comparisons. There is nothing to prevent mergesort from taking less time on some other input of length n.

The exact interpretation of statements involving O, o, and Ω depends on assumptions about the underlying model of computation, how the input is presented, how the size of the input is determined, and what constitutes a single step of the computation. In practice, authors often do not bother to write these down. For example, "The running time of mergesort is $O(n \log n)$" means that there is a fixed constant c such that for any n elements drawn from a totally ordered set, at most $cn \log n$ comparisons are needed to produce a sorted array. Here nothing is counted in the running time except the number of comparisons between individual elements, and each comparison is assumed to take one step; other operations are ignored. Similarly, nothing is counted in the input size except the number of elements; the size of each element (whatever that may mean) is ignored.

It is important to be aware of these unstated assumptions and understand how to make them explicit and formal when reading papers in the field. When making such statements yourself, always have your underlying assumptions in mind. Although many authors don't bother, it is a good habit to state any assumptions about the model of computation explicitly in any papers you write.

The question of what assumptions are reasonable is more often than not a matter of esthetics. You will become familiar with the standard models and assumptions from reading the literature; beyond that, you must depend on your own conscience.

1.2 Models of Computation

Our principal model of computation will be the unit-cost random access machine (RAM). Other models, such as uniform circuits and PRAMs, will be introduced when needed. The RAM model allows random access and the use

of arrays, as well as unit-cost arithmetic and bit-vector operations on arbitrarily large integers; see [3].

For graph algorithms, arithmetic is often unnecessary. Of the two main representations of graphs, namely *adjacency matrices* and *adjacency lists*, the former requires random access and $\Omega(n^2)$ array storage; the latter, only linear storage and no random access. (For graphs, *linear* means $O(n+m)$, where n is the number of vertices of the graph and m is the number of edges.) The most esthetically pure graph algorithms are those that use the adjacency list representation and only manipulate pointers. To express such algorithms one can formulate a very weak model of computation with primitive operators equivalent to **car**, **cdr**, **cons**, **eq**, and **nil** of pure LISP; see also [99].

1.3 A Grain of Salt

No mathematical model can reflect reality with perfect accuracy. Mathematical models are abstractions; as such, they are necessarily flawed.

For example, it is well known that it is possible to abuse the power of unit-cost RAMs by encoding horrendously complicated computations in large integers and solving intractible problems in polynomial time [50]. However, this violates the unwritten rules of good taste. One possible preventative measure is to use the log-cost model; but when used as intended, the unit-cost model reflects experimental observation more accurately for data of moderate size (since multiplication really does take one unit of time), besides making the mathematical analysis a lot simpler.

Some theoreticians consider asymptotically optimal results as a kind of Holy Grail, and pursue them with a relentless frenzy (present company not necessarily excluded). This often leads to contrived and arcane solutions that may be superior by the measure of asymptotic complexity, but whose constants are so large or whose implementation would be so cumbersome that no improvement in technology would ever make them feasible. What is the value of such results? Sometimes they give rise to new data structures or new techniques of analysis that are useful over a range of problems, but more often than not they are of strictly mathematical interest. Some practitioners take this activity as an indictment of asymptotic complexity itself and refuse to admit that asymptotics have anything at all to say of interest in practical software engineering.

Nowhere is the argument more vociferous than in the theory of parallel computation. There are those who argue that many of the models of computation in common use, such as uniform circuits and PRAMs, are so inaccurate as to render theoretical results useless. We will return to this controversy later on when we talk about parallel machine models.

Such extreme attitudes on either side are unfortunate and counterproductive. By now asymptotic complexity occupies an unshakable position in our computer science consciousness, and has probably done more to guide us in

improving technology in the design and analysis of algorithms than any other mathematical abstraction. On the other hand, one should be aware of its limitations and realize that an asymptotically optimal solution is not necessarily the best one.

A good rule of thumb in the design and analysis of algorithms, as in life, is to use common sense, exercise good taste, and always listen to your conscience.

1.4 Strassen's Matrix Multiplication Algorithm

Probably the single most important technique in the design of asymptotically fast algorithms is *divide-and-conquer*. Just to refresh our understanding of this technique and the use of recurrences in the analysis of algorithms, let's take a look at Strassen's classical algorithm for matrix multiplication and some of its progeny. Some of these examples will also illustrate the questionable lengths to which asymptotic analysis can sometimes be taken.

The usual method of matrix multiplication takes 8 multiplications and 4 additions to multiply two 2×2 matrices, or in general $O(n^3)$ arithmetic operations to multiply two $n \times n$ matrices. However, the number of multiplications can be reduced. Strassen [97] published one such algorithm for multiplying 2×2 matrices using only 7 multiplications and 18 additions:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} s_1 + s_2 - s_4 + s_6 & s_4 + s_5 \\ s_6 + s_7 & s_2 - s_3 + s_5 - s_7 \end{bmatrix}$$

where

$$\begin{aligned} s_1 &= (b-d)\cdot(g+h) \\ s_2 &= (a+d)\cdot(e+h) \\ s_3 &= (a-c)\cdot(e+f) \\ s_4 &= h\cdot(a+b) \\ s_5 &= a\cdot(f-h) \\ s_6 &= d\cdot(g-e) \\ s_7 &= e\cdot(c+d)\ . \end{aligned}$$

Assume for simplicity that n is a power of 2. (This is not the last time you will hear that.) Apply the 2×2 algorithm recursively on a pair of $n \times n$ matrices by breaking each of them up into four square submatrices of size $\frac{n}{2} \times \frac{n}{2}$:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \cdot \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} S_1 + S_2 - S_4 + S_6 & S_4 + S_5 \\ S_6 + S_7 & S_2 - S_3 + S_5 - S_7 \end{bmatrix}$$

where

$$S_1 = (B-D)\cdot(G+H)$$

$$\begin{aligned} S_2 &= (A+D)\cdot(E+H) \\ S_3 &= (A-C)\cdot(E+F) \\ S_4 &= H\cdot(A+B) \\ S_5 &= A\cdot(F-H) \\ S_6 &= D\cdot(G-E) \\ S_7 &= E\cdot(C+D)\ . \end{aligned}$$

Everything is the same as in the 2×2 case, except now we are manipulating $\frac{n}{2} \times \frac{n}{2}$ matrices instead of scalars. (We have to be slightly cautious, since matrix multiplication is not commutative.) Ultimately, how many scalar operations $(+, -, \cdot)$ does this recursive algorithm perform in multiplying two $n \times n$ matrices? We get the recurrence

$$T(n) = 7T(\frac{n}{2}) + dn^2$$

with solution

$$\begin{aligned} T(n) &= (1+\frac{4}{3}d)n^{\log_2 7} + O(n^2) \\ &= O(n^{\log_2 7}) \\ &= O(n^{2.81\ldots}) \end{aligned}$$

which is $o(n^3)$. Here d is a fixed constant, and dn^2 represents the time for the matrix additions and subtractions.

This is already a significant asymptotic improvement over the naive algorithm, but can we do even better? In general, an algorithm that uses c multiplications to multiply two $d \times d$ matrices, used as the basis of such a recursive algorithm, will yield an $O(n^{\log_d c})$ algorithm. To beat Strassen's algorithm, we must have $c < d^{\log_2 7}$. For a 3×3 matrix, we need $c < 3^{\log_2 7} = 21.8\ldots$, but the best known algorithm uses 23 multiplications.

In 1978, Victor Pan [83, 84] showed how to multiply 70×70 matrices using 143640 multiplications. This gives an algorithm of approximately $O(n^{2.795\ldots})$. The asymptotically best algorithm known to date, which is achieved by entirely different methods, is $O(n^{2.376\ldots})$ [25]. Every algorithm must be $\Omega(n^2)$, since it has to look at all the entries of the matrices; no better lower bound is known.

Lecture 2 Topological Sort and MST

A recurring theme in asymptotic analysis is that it is often possible to get better asymptotic performance by maintaining extra information about the structure. Updating this extra information may slow down each individual step; this additional cost is sometimes called *overhead*. However, it is often the case that a small amount of overhead yields dramatic improvements in the asymptotic complexity of the algorithm.

To illustrate, let's look at *topological sort*. Let $G = (V, E)$ be a directed acyclic graph (dag). The edge set E of the dag G induces a *partial order* (a reflexive, antisymmetric, transitive binary relation) on V, which we denote by E^* and define by: uE^*v if there exists a directed E-path of length 0 or greater from u to v. The relation E^* is called the *reflexive transitive closure* of E.

Proposition 2.1 *Every partial order extends to a total order (a partial order in which every pair of elements is comparable).*

Proof. If R is a partial order that is not a total order, then there exist u, v such that neither uRv nor vRu. Extend R by setting

$$R \;:=\; R \cup \{(x, y) \mid xRu \text{ and } vRy\} \; .$$

The new R is a partial order extending the old R, and in addition now uRv. Repeat until there are no more incomparable pairs. □

In the case of a dag $G = (V, E)$ with associated partial order E^*, to say that a total order $\leq$ extends E^* is the same as saying that if uEv then $u \leq v$. Such a total order is called a *topological sort* of the dag G. A naive $O(n^3)$ algorithm to find a topological sort can be obtained from the proof of the above proposition.

Here is a faster algorithm, although still not optimal.

Algorithm 2.2 (Topological Sort II)

1. Start from any vertex and follow edges backwards until finding a vertex u with no incoming edges. Such a u must be encountered eventually, since there are no cycles and the dag is finite.
2. Make u the next vertex in the total order.
3. Delete u and all adjacent edges and go to step 1.

Using the adjacency list representation, the running time of this algorithm is $O(n)$ steps per iteration for n iterations, or $O(n^2)$.

The bottleneck here is step 1. A minor modification will allow us to perform this step in constant time. Assume the adjacency list representation of the graph associates with each vertex two separate lists, one for the incoming edges and one for the outgoing edges. If the representation is not already of this form, it can easily be put into this form in linear time. The algorithm will maintain a queue of vertices with no incoming edges. This will reduce the cost of finding a vertex with no incoming edges to constant time at a slight extra overhead for maintaining the queue.

Algorithm 2.3 (Topological Sort III)

1. Initialize the queue by traversing the graph and inserting each v whose list of incoming edges is empty.
2. Pick a vertex u off the queue and make u the next vertex in the total order.
3. Delete u and all outgoing edges (u, v). For each such v, if its list of incoming edges becomes empty, put v on the queue. Go to step 2.

Step 1 takes time $O(n)$. Step 2 takes constant time, thus $O(n)$ time over all iterations. Step 3 takes time $O(m)$ over all iterations, since each edge can be deleted at most once. The overall time is $O(m + n)$.

Later we will see a different approach involving depth first search.

2.1 Minimum Spanning Trees

Let $G = (V, E)$ be a connected undirected graph.

Definition 2.4 A *forest* in G is a subgraph $F = (V, E')$ with no cycles. Note that F has the same vertex set as G. A *spanning tree* in G is a forest with exactly one connected component. Given weights $w : E \to \mathcal{N}$ (edges are assigned weights over the natural numbers), a *minimum (weight) spanning tree (MST)* in G is a spanning tree T whose total weight (sum of the weights of the edges in T) is minimum over all spanning trees. □

Lemma 2.5 *Let $F = (V, E)$ be an undirected graph, c the number of connected components of F, $m = |E|$, and $n = |V|$. Then F has no cycles iff $c + m = n$.*

Proof.
($\rightarrow$) By induction on m. If $m = 0$, then there are n vertices and each forms a connected component, so $c = n$. If an edge is added without forming a cycle, then it must join two components. Thus m is increased by 1 and c is decreased by 1, so the equation $c + m = n$ is maintained.

($\leftarrow$) Suppose that F has at least one cycle. Pick an arbitrary cycle and remove an edge from that cycle. Then m decreases by 1, but c and n remain the same. Repeat until there are no more cycles. When done, the equation $c + m = n$ holds, by the preceding paragraph; but then it could not have held originally. □

We use a *greedy algorithm* to produce a minimum weight spanning tree. This algorithm is originally due to Kruskal [66].

Algorithm 2.6 (Greedy Algorithm for MST)

1. Sort the edges by weight.
2. For each edge on the list in order of increasing weight, include that edge in the spanning tree if it does not form a cycle with the edges already taken; otherwise discard it.

The algorithm can be halted as soon as $n - 1$ edges have been kept, since we know we have a spanning tree by Lemma 2.5.

Step 1 takes time $O(m \log m) = O(m \log n)$ using any one of a number of general sorting methods, but can be done faster in certain cases, for example if the weights are small integers so that bucket sort can be used.

Later on, we will give an almost linear time implementation of step 2, but for now we will settle for $O(n \log n)$. We will think of including an edge e in the spanning tree as taking the union of two disjoint sets of vertices, namely the vertices in the connected components of the two endpoints of e in the forest

being built. We represent each connected component as a linked list. Each list element points to the next element and has a back pointer to the head of the list. Initially there are no edges, so we have n lists, each containing one vertex. When a new edge (u, v) is encountered, we check whether it would form a cycle, *i.e.* whether u and v are in the same connected component, by comparing back pointers to see if u and v are on the same list. If not, we add (u, v) to the spanning tree and take the union of the two connected components by merging the two lists. Note that the lists are always disjoint, so we don't have to check for duplicates.

Checking whether u and v are in the same connected component takes constant time. Each merge of two lists could take as much as linear time, since we have to traverse one list and change the back pointers, and there are $n - 1$ merges; this will give $O(n^2)$ if we are not careful. However, if we maintain counters containing the size of each component and always merge the smaller into the larger, then each vertex can have its back pointer changed at most $\log n$ times, since each time the size of its component at least doubles. If we charge the change of a back pointer to the vertex itself, then there are at most $\log n$ changes per vertex, or at most $n \log n$ in all. Thus the total time for all list merges is $O(n \log n)$.

2.2 The Blue and Red Rules

Here is a more general approach encompassing most of the known algorithms for the MST problem. For details and references, see [100, Chapter 6], which proves the correctness of the greedy algorithm as a special case of this more general approach. In the next lecture, we will give an even more general treatment.

Let $G = (V, E)$ be an undirected connected graph with edge weights $w : E \to \mathcal{N}$. Consider the following two rules for coloring the edges of G, which Tarjan [100] calls the *blue rule* and the *red rule*:

> **Blue Rule:** Find a *cut* (a partition of V into two disjoint sets X and $V - X$) such that no blue edge crosses the cut. Pick an uncolored edge of minimum weight between X and $V - X$ and color it blue.
> **Red Rule:** Find a *cycle* (a path in G starting and ending at the same vertex) containing no red edge. Pick an uncolored edge of maximum weight on that cycle and color it red.

The greedy algorithm is just a repeated application of a special case of the blue rule. We will show next time:

Theorem 2.7 *Starting with all edges uncolored, if the blue and red rules are applied in arbitrary order until neither applies, then the final set of blue edges forms a minimum spanning tree.*

Lecture 3 Matroids and Independence

Before we prove the correctness of the blue and red rules for MST, let's first discuss an abstract combinatorial structure called a *matroid*. We will show that the MST problem is a special case of the more general problem of finding a minimum-weight maximal independent set in a matroid. We will then generalize the blue and red rules to arbitrary matroids and prove their correctness in this more general setting. We will show that every matroid has a dual matroid, and that the blue and red rules of a matroid are the red and blue rules, respectively, of its dual. Thus, once we establish the correctness of the blue rule, we get the red rule for free.

We will also show that a structure is a matroid if and only if the greedy algorithm always produces a minimum-weight maximal independent set for any weighting.

Definition 3.1 A *matroid* is a pair $(S, \mathcal{I})$ where S is a finite set and $\mathcal{I}$ is a family of subsets of S such that

(i) if $J \in \mathcal{I}$ and $I \subseteq J$, then $I \in \mathcal{I}$;

(ii) if $I, J \in \mathcal{I}$ and $|I| < |J|$, then there exists an $x \in J - I$ such that $I \cup \{x\} \in \mathcal{I}$.

The elements of $\mathcal{I}$ are called *independent sets* and the subsets of S not in $\mathcal{I}$ are called *dependent sets*. □

This definition is supposed to capture the notion of *independence* in a general way. Here are some examples:

1. Let V be a vector space, let S be a finite subset of V, and let $\mathcal{I} \subseteq 2^S$ be the family of linearly independent subsets of S. This example justifies the term "independent".

2. Let A be a matrix over a field, let S be the set of rows of A, and let $\mathcal{I} \subseteq 2^S$ be the family of linearly independent subsets of S.

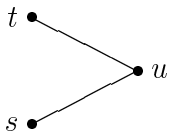
3. Let $G = (V, E)$ be a connected undirected graph. Let $S = E$ and let $\mathcal{I}$ be the set of forests in G. This example gives the MST problem of the previous lecture.

4. Let $G = (V, E)$ be a connected undirected graph. Let $S = E$ and let $\mathcal{I}$ be the set of subsets $E' \subseteq E$ such that the graph $(V, E - E')$ is connected.

5. Elements $\alpha_1, \ldots, \alpha_n$ of a field are said to be *algebraically independent* over a subfield k if there is no nontrivial polynomial $p(x_1, \ldots, x_n)$ with coefficients in k such that $p(\alpha_1, \ldots, \alpha_n) = 0$. Let S be a finite set of elements and let $\mathcal{I}$ be the set of subsets of S that are algebraically independent over k.

Definition 3.2 A *cycle* (or *circuit*) of a matroid $(S, \mathcal{I})$ is a setwise minimal (*i.e.*, minimal with respect to set inclusion) dependent set. A *cut* (or *cocircuit*) of $(S, \mathcal{I})$ is a setwise minimal subset of S intersecting all maximal independent sets. □

The terms *circuit* and *cocircuit* are standard in matroid theory, but we will continue to use *cycle* and *cut* to maintain the intuitive connection with the special case of MST. However, be advised that cuts in graphs as defined in the last lecture are *unions* of cuts as defined here. For example, in the graph



the set $\{(s, u),\ (t, u)\}$ forms a cut in the sense of MST, but not a cut in the sense of the matroid, because it is not minimal. However, a moment's thought reveals that this difference is inconsequential as far as the blue rule is concerned.

Let the elements of S be weighted. We wish to find a setwise maximal independent set whose total weight is minimum among all setwise maximal independent sets. In this more general setting, the blue and red rules become:

Blue Rule: Find a cut with no blue element. Pick an uncolored element of the cut of minimum weight and color it blue.
Red Rule: Find a cycle with no red element. Pick an element of the cycle of maximum weight and color it red.

3.1 Matroid Duality

As the astute reader has probably noticed by now, there is some kind of duality afoot. The similarity between the blue and red rules is just too striking to be mere coincidence.

Definition 3.3 Let $(S, \mathcal{I})$ be a matroid. The *dual matroid* of $(S, \mathcal{I})$ is $(S, \mathcal{I}^*)$, where

$$\mathcal{I}^* = \{\text{subsets of } S \text{ disjoint from some maximal element of } \mathcal{I}\} .$$

In other words, the maximal elements of $\mathcal{I}^*$ are the complements in S of the maximal elements of $\mathcal{I}$. □

The examples 3 and 4 above are duals. Note that $\mathcal{I}^{**} = \mathcal{I}$. Be careful: it is *not* the case that a set is independent in a matroid iff it is dependent in its dual. For example, except in trivial cases, $\emptyset$ is independent in both matroids.

Theorem 3.4

1. *Cuts in $(S, \mathcal{I})$ are cycles in $(S, \mathcal{I}^*)$.*
2. *The blue rule in $(S, \mathcal{I})$ is the red rule in $(S, \mathcal{I}^*)$ with the ordering of the weights reversed.*

3.2 Correctness of the Blue and Red Rules

Now we prove the correctness of the blue and red rules in arbitrary matroids. A proof for the special case of MST can be found in Tarjan's book [100, Chapter 6]; Lawler [70] states the blue and red rules for arbitrary matroids but omits a proof of correctness.

Definition 3.5 Let $(S, \mathcal{I})$ be a matroid with dual $(S, \mathcal{I}^*)$. An *acceptable coloring* is a pair of disjoint sets $B \in \mathcal{I}$ (the *blue elements*) and $R \in \mathcal{I}^*$ (the *red elements*). An acceptable coloring B, R is *total* if $B \cup R = S$, *i.e.* if B is a maximal independent set and R is a maximal independent set in the dual. An acceptable coloring B', R' *extends* or *is an extension of* an acceptable coloring B, R if $B \subseteq B'$ and $R \subseteq R'$. □

Lemma 3.6 *Any acceptable coloring has a total acceptable extension.*

Proof. Let B, R be an acceptable coloring. Let U^* be a maximal element of $\mathcal{I}^*$ extending R, and let $U = S - U^*$. Then U is a maximal element of $\mathcal{I}$ disjoint from R. As long as $|B| < |U|$, select elements of U and add them to B, maintaining independence. This is possible by axiom (ii) of matroids. Let $\widehat{B}$ be the resulting set. Since all maximal independent sets have the same cardinality (Exercise 1a, Homework 1), $\widehat{B}$ is a maximal element of $\mathcal{I}$ containing B and disjoint from R. The desired total extension is $\widehat{B}, S - \widehat{B}$. □

Lemma 3.7 *A cut and a cycle cannot intersect in exactly one element.*

Proof. Let C be a cut and D a cycle. Suppose that $C \cap D = \{x\}$. Then $D - \{x\}$ is independent and $C - \{x\}$ is independent in the dual. Color $D - \{x\}$ blue and $C - \{x\}$ red; by Lemma 3.6, this coloring extends to a total acceptable coloring. But depending on the color of x, either C is all red or D is all blue; this is impossible in an acceptable coloring, since D is dependent and C is dependent in the dual. □

Suppose B is independent and $B \cup \{x\}$ is dependent. Then $B \cup \{x\}$ contains a minimal dependent subset or cycle C, called the *fundamental cycle*[1] of x and B. The cycle C must contain x, because $C - \{x\}$ is contained in B and is therefore independent.

Lemma 3.8 (Exchange Lemma) *Let B, R be a total acceptable coloring.*

(i) Let $x \in R$ and let y lie on the fundamental cycle of x and B. If the colors of x and y are exchanged, the resulting coloring is acceptable.

(ii) Let $y \in B$ and let x lie on the fundamental cut of y and R (the fundamental cut of y and R is the fundamental cycle of y and R in the dual matroid). If the colors of x and y are exchanged, the resulting coloring is acceptable.

Proof. By duality, we need only prove (i). Let C be the fundamental cycle of x and B and let y lie on C. If $y = x$, there is nothing to prove. Otherwise $y \in B$. The set $C - \{y\}$ is independent since C is minimal. Extend $C - \{y\}$ by adding elements of $|B|$ as in the proof of Lemma 3.6 until achieving a maximal independent set B'. Then $B' = (B - \{y\}) \cup \{x\}$, and the total acceptable coloring $B', S - B'$ is obtained from B, R by switching the colors of x and y. □

A total acceptable coloring B, R is called *optimal* if B is of minimum weight among all maximal independent sets; equivalently, if R is of maximum weight among all maximal independent sets in the dual matroid.

Lemma 3.9 *If an acceptable coloring has an optimal total extension before execution of the blue or red rule, then so has the resulting coloring afterwards.*

Proof. We prove the case of the blue rule; the red rule follows by duality. Let B, R be an acceptable coloring with optimal total extension $\widehat{B}, \widehat{R}$. Let A be a cut containing no blue elements, and let x be an uncolored element of A of minimum weight. If $x \in \widehat{B}$, we are done, so assume that $x \in \widehat{R}$. Let C be the fundamental cycle of x and $\widehat{B}$. By Lemma 3.7, $A \cap C$ must contain

[1] We say "the" because it is unique (Exercise 1b, Homework 1), although we do not need to know this for our argument.

another element besides x, say y. Then $y \in \widehat{B}$, and $y \notin B$ because there are no blue elements of A. By Lemma 3.8, the colors of x and y in $\widehat{B}, \widehat{R}$ can be exchanged to obtain a total acceptable coloring $\widehat{B}', \widehat{R}'$ extending $B \cup \{x\}, R$. Moreover, $\widehat{B}'$ is of minimum weight, because the weight of x is no more than that of y. □

We also need to know

Lemma 3.10 *If an acceptable coloring is not total, then either the blue or red rule applies.*

Proof. Let B, R be an acceptable coloring with uncolored element x. By Lemma 3.6, B, R has a total extension $\widehat{B}, \widehat{R}$. By duality, assume without loss of generality that $x \in \widehat{B}$. Let C be the fundamental cut of x and $\widehat{R}$. Since all elements of C besides x are in $\widehat{R}$, none of them are blue in B. Thus the blue rule applies. □

Combining Lemmas 3.9 and 3.10, we have

Theorem 3.11 *If we start with an uncolored weighted matroid and apply the blue or red rules in any order until neither applies, then the resulting coloring is an optimal total acceptable coloring.*

What is really going on here is that all the subsets of the maximal independent sets of minimal weight form a submatroid of $(S, \mathcal{I})$, and the blue rule gives a method for implementing axiom (ii) for this matroid; see Miscellaneous Exercise 1.

3.3 Matroids and the Greedy Algorithm

We have shown that if $(S, \mathcal{I})$ is a matroid, then the greedy algorithm produces a maximal independent set of minimum weight. Here we show the converse: if $(S, \mathcal{I})$ is not a matroid, then the greedy algorithm fails for some choice of integer weights. Thus the abstract concept of matroid captures exactly when the greedy algorithm works.

Theorem 3.12 ([32]; see also [70]) *A system $(S, \mathcal{I})$ satisfying axiom (i) of matroids is a matroid (i.e., it satisfies (ii)) if and only if for all weight assignments $w : S \to \mathcal{N}$, the greedy algorithm gives a minimum-weight maximal independent set.*

Proof. The direction ($\rightarrow$) has already been shown. For ($\leftarrow$), let $(S, \mathcal{I})$ satisfy (i) but not (ii). There must be A, B such that $A, B \in \mathcal{I}$, $|A| < |B|$, but for no $x \in B - A$ is $A \cup \{x\} \in \mathcal{I}$.

Assume without loss of generality that B is a *maximal* independent set. If it is not, we can add elements to B maintaining the independence of B; for

any element that we add to B that can also be added to A while preserving the independence of A, we do so. This process never changes the fact that $|A| < |B|$ and for no $x \in B - A$ is $A \cup \{x\} \in \mathcal{I}$.

Now we assign weights $w : S \to \mathcal{N}$. Let $a = |A - B|$ and $b = |B - A|$. Then $a < b$. Let h be a huge number, $h \gg a, b$. (Actually $h > b^2$ will do.)

Case 1 If A is a maximal independent set, assign

$$\begin{aligned} w(x) &= a + 1 \quad \text{for } x \in B - A \\ w(x) &= b + 1 \quad \text{for } x \in A - B \\ w(x) &= 0 \qquad \text{for } x \in A \cap B \\ w(x) &= h \qquad \text{for } x \notin A \cup B \ . \end{aligned}$$

Thus

$$\begin{aligned} w(A) &= a(b+1) = ab + a \\ w(B) &= b(a+1) = ab + b \ . \end{aligned}$$

This weight assignment forces the greedy algorithm to choose B when in fact A is a maximal independent set of smaller weight.

Case 2 If A is not a maximal independent set, assign

$$\begin{aligned} w(x) &= 0 \quad \text{for } x \in A \\ w(x) &= b \quad \text{for } x \in B - A \\ w(x) &= h \quad \text{for } x \notin A \cup B \ . \end{aligned}$$

All the elements of A will be chosen first, and then a huge element outside of $A \cup B$ must be chosen, since A is not maximal. Thus the minimum-weight maximal independent set B was not chosen. □

Lecture 4 Depth-First and Breadth-First Search

Depth-first search (DFS) and breadth-first search (BFS) are two of the most useful subroutines in graph algorithms. They allow one to search a graph in linear time and compile information about the graph. They differ in that the former uses a stack (LIFO) discipline and the latter uses a queue (FIFO) discipline to choose the next edge to explore.

Undirected depth-first search produces in linear time a numbering of the vertices called the *depth-first numbering* and a particular spanning tree called the *depth-first spanning tree* of each connected component. This is done as follows. Choose an arbitrary vertex u, which will become the root of the tree. Push all edges $(u, v) \in E$ onto the stack. Assign u the DFS number 0 and set the DFS counter c to 1. Now repeat the following activity until the stack becomes empty. Let (x, y) be the top element of the stack. This is the next edge to explore. The vertex x has a DFS number already (this is an invariant of the loop). If y has no DFS number, assign it the DFS number c, increment c, push all edges $(y, z) \in E$ onto the stack, and make the (directed) edge (x, y) a *tree edge*. Otherwise, if y has a DFS number already, just pop (x, y) off the stack.

The tree edges form a directed spanning tree of the connected component of u rooted at u. It is a dag rooted at u, since tree edges (x, y) only go from lower numbered vertices to higher numbered vertices. It is a tree, since no vertex has indegree greater than one; this is because (x, y) becomes a tree edge only if y has no DFS number, and thereafter y has a DFS number. It is

a spanning tree, since it is easily shown inductively that every vertex in the connected component of u eventually receives a DFS number. This spanning tree is called the *depth-first spanning tree*.

We can repeat the whole process with a new arbitrarily chosen unvisited vertex to search the other connected components.

The non-tree edges (x, y) are called *back edges* and are directed from higher numbered to lower numbered vertices. When we draw a DFS tree, we usually draw the root at the top, the tree edges pointing down (hence the term *depth-first*), and the back edges pointing up.

Back edges out of v can only go to ancestors of v in the DFS tree. There cannot be a back edge to a nonancestor, since that edge would have been explored earlier from the other direction and would have been a tree edge.

DFS takes time $O(m + n)$ where n is the number of vertices and m is the number of edges, since each edge is stacked at most once in each direction, and each edge and vertex requires a constant amount of processing.

See [3, 78] for an alternative treatment.

4.1 Biconnected Components

Let $G = (V, E)$ be a connected undirected graph.

Definition 4.1 A vertex v is an *articulation point* if its removal disconnects the graph. □

Definition 4.2 A connected graph is called *biconnected* if any pair of distinct vertices u and v lie on a simple cycle (one with no repeated vertices). □

Note that according to this definition, a graph with two vertices connected by a single edge is biconnected (no one said anything about not repeating edges).

If G is not biconnected, we define the *biconnected components* of G in terms of an equivalence relation on edges:

Definition 4.3 For $e, e' \in E$, define $e \equiv e'$ if e and e' lie on a simple cycle. □

Lemma 4.4 *The relation $\equiv$ is an equivalence relation (reflexive, symmetric, and transitive).*

Proof. Reflexivity $e \equiv e$ follows from the fact that the edge e and its two endpoints constitute a simple cycle. The relation is symmetric, since e and e' can be interchanged in the definition of $\equiv$. The hard one is transitivity. Suppose $(u, v) \equiv (u', v')$ and $(u', v') \equiv (u'', v'')$. Let c and c' be the two simple cycles involved, respectively. Assume u, u', v', v occur in that order around c. Let x be the first vertex on the segment of c from u to u' that also lies in c'; x must exist since $u' \in c'$, at least. Let y be the first vertex on the segment of

c from v to v' that also lies in c'; y must exist since $v' \in c'$. Also, $x \neq y$ since c is simple. Let p be the path from x to y in c containing (u, v) and let p' be the path from x to y in c' containing (u'', v''). Then p and p' intersect only in x and y, and together form a simple cycle containing (u, v) and (u'', v''). □

Definition 4.5 The equivalence classes of $\equiv$ are called *biconnected components.* □

Lemma 4.6 *The vertex a is an articulation point iff a is contained in at least two biconnected components.*

Proof. Suppose the removal of a disconnects the graph. Then there exist u and v adjacent to a such that every path from u to v goes through a. Then the edges (u, a) and (a, v) cannot lie on a simple cycle, thus are in different biconnected components.

Conversely, suppose u and v are adjacent to a and $(u, a) \not\equiv (a, v)$. Then all paths between u and v must go through a. Thus if a is removed, there is no path between u and v, so G is disconnected. □

Below, when using the terms "descendant" and "ancestor" in a depth-first search tree, we will always consider a vertex u to be a descendant of itself and an ancestor of itself. In other words, we take the descendant and ancestor relations to be reflexive. If we want to exclude u, we do so explicitly by using the terms "proper descendant" and "proper ancestor".

Lemma 4.7 *Let (u, v) and (v, w) be two adjacent tree edges in a depth-first search tree of G. Then $(u, v) \equiv (v, w)$ if and only if there exists a back edge from some descendant of w to some ancestor of u.*

Proof.

($\rightarrow$) If there exists a back edge from some descendant w' of w to some ancestor u' of u, then (u, v) and (v, w) are edges in a simple cycle consisting of the back edge (w', u') along with the path of tree edges from u' to w'. Thus $(u, v) \equiv (v, w)$.

($\leftarrow$) Suppose $(u, v) \equiv (v, w)$. Then there must be a simple cycle containing them. This cycle must contain the edges (u, v) and (v, w) in this order, since it may only go through v once. Consider the subtree of the depth-first tree rooted at w. The simple cycle must contain a back edge (w', u') out of this subtree, since it must get back to u eventually. (Before coming out, the path inside the subtree can be quite complicated, since it can traverse tree and back edges in either direction—don't forget that the graph is undirected.) Then w' is a descendant of w and u' is an ancestor of w'. Since u' is not in the subtree rooted at w, it must be an ancestor of v. But it cannot be v because v cannot be used twice on the cycle. Therefore u' must be an ancestor of u. □

The biconnected components can be found from a DFS tree as follows. Assume the vertices are named by their DFS numbers. We compute a value for each vertex v, called $\mathbf{low}(v)$, which gives the DFS number of the lowest numbered vertex x (*i.e.* the highest in the tree) such that there is a back edge from some descendant of v to x. By Lemmas 4.6 and 4.7, a vertex u will be an articulation point, and the biconnected component of the tree edge (u, v) will lie entirely in the subtree rooted at u, if $\mathbf{low}(v) \geq u$. We can inductively compute $\mathbf{low}(v)$ as follows:

$$\begin{aligned} x &:= \min\{\mathbf{low}(w) \mid w \text{ is an immediate descendant of } v\} \\ y &:= \min\{z \mid z \text{ is reachable by a back edge from } v\} \\ \mathbf{low}(v) &:= \min(x, y) \ . \end{aligned}$$

The values $\mathbf{low}(v)$ can be computed simultaneously with the construction of the DFS tree in linear time. As soon as an articulation point u is discovered with (u, v) a tree edge such that $\mathbf{low}(v) \geq u$, the biconnected component containing the edge (u, v) can be deleted from the graph. See [3, 78] for more details.

4.2 Directed DFS

The DFS procedure on directed graphs is similar to DFS on undirected graphs, except that we only follow edges from sources to sinks. Four types of edges can result:

- *tree edges* to a vertex not yet visited
- *back edges* to an ancestor
- *forward edges* to a descendant previously visited
- *cross edges* to a vertex previously visited that is neither an ancestor nor a descendant.

There can be no cross edges to a higher numbered vertex; such an edge would have been a tree edge. If we mark the vertex y when the tree edge (x, y) is popped to indicate that the subtree below y has been completely explored, we can recognize each of these four cases when we explore the edge (u, v) by checking marks and comparing DFS numbers:

(u, v) *is a*	*if*
tree edge	DFS(v) does not exist
back edge	DFS(v) < DFS(u) and v is not marked
forward edge	DFS(v) > DFS(u)
cross edge	DFS(v) < DFS(u) and v is marked

The directed DFS tree can be constructed in linear time; see [3, 78] for details.

The first application of directed DFS is determining acyclicity:

Theorem 4.8 *A directed graph is acyclic iff its DFS forest has no back edges.*

Proof. If there is a back edge, the graph is surely cyclic. Conversely, if there are no back edges, consider the *postorder* numbering of the DFS forest: traverse the forest in depth-first order, but number the vertices in the order they are *last* seen. Then tree edges, forward edges, and cross edges all go from higher numbered to lower numbered vertices, so there can be no cycles. □

4.3 Strong Components

Definition 4.9 Let $G = (V, E)$ be a directed graph. For $u, v \in V$, define $u \equiv v$ if u and v lie on a directed cycle in G. This is an equivalence relation, and its equivalence classes are called *strongly connected components* or just *strong components*. A graph G is said to be *strongly connected* if for any pair of vertices u, v there is a directed cycle in G containing u and v; *i.e.*, if G has only one strong component. □

The strong components of a directed graph can be computed in linear time using directed depth-first search. The algorithm is similar to the algorithm for biconnected components in undirected graphs; see [3] for details.

4.4 Strong Components and Partial Orders

Strong components are important in the representation of partial orders. Finite partial orders are often represented as the reflexive transitive closures E^* of dags $G = (V, E)$ (recall $(u, v) \in E^*$ iff there exists an E-path from u to v of length 0 or greater). If G is not acyclic, then the relation E^* does not satisfy the antisymmetry law, and is thus not a partial order. However, it is still reflexive and transitive. Such a relation is called a *preorder* or sometimes a *quasiorder*.

Given an arbitrary preorder $(P, \preceq)$, define $x \approx y$ if $x \preceq y$ and $y \preceq x$. This is an equivalence relation, and we can collapse its equivalence classes into single points to get a partial order. This construction is called a *quotient construction*. Formally, let $[x]$ denote the $\approx$-class of x and let $P/\!\approx$ denote the set of all such classes; *i.e.*,

$$\begin{aligned} [x] &= \{y \mid y \approx x\} \\ P/\!\approx &= \{[x] \mid x \in P\} \ . \end{aligned}$$

The preorder $\preceq$ induces a preorder, also denoted $\preceq$, on $P/\!\approx$ in a natural way: $[x] \preceq [y]$ if $x \preceq y$ in P. (The choice of x and y in their respective equivalence classes doesn't matter.) It is easily shown that the preorder $\preceq$ is

actually a partial order on $P/\approx$; intuitively, by collapsing equivalence classes, we identified those elements that caused antisymmetry to fail.

Forming the strong components of a directed (not necessarily acyclic) graph $G = (V, E)$ allows us to perform this operation effectively on the preorder (V, E^*). We form a quotient graph $G/\equiv$ by collapsing the strong components of G into single vertices:

$$\begin{aligned} [v] &= \{u \mid u \equiv v\} \text{ (the strong component of } v) \\ V/\equiv &= \{[v] \mid v \in V\} \\ E' &= \{([u], [v]) \mid (u, v) \in E\} \\ G/\equiv &= (V/\equiv, E') \ . \end{aligned}$$

It is not hard to show that $G/\equiv$ is acyclic. Moreover,

Theorem 4.10 *The partial orders $(V/\approx, E^*)$ and $(V/\equiv, (E')^*)$ are isomorphic.*

In other words, the partial order represented by the collapsed graph is the same as the collapse of the preorder represented by the original graph.

Lecture 5 Shortest Paths and Transitive Closure

5.1 Single-Source Shortest Paths

Let $G = (V, E)$ be an undirected graph and let ℓ be a function assigning a nonnegative length to each edge. Extend ℓ to domain $V \times V$ by defining $\ell(v, v) = 0$ and $\ell(u, v) = \infty$ if $(u, v) \notin E$. Define the *length*[2] of a path $p = e_1 e_2 \ldots e_n$ to be $\ell(p) = \sum_{i=1}^{n} \ell(e_i)$. For $u, v \in V$, define the *distance* $d(u, v)$ from u to v to be the length of a shortest path from u to v, or ∞ if no such path exists. The *single-source shortest path problem* is to find, given $s \in V$, the value of $d(s, u)$ for every other vertex u in the graph.

If the graph is unweighted (*i.e.*, all edge lengths are 1), we can solve the problem in linear time using BFS. For the more general case, here is an algorithm due to Dijkstra [28]. Later on we will give an $O(m + n \log n)$ implementation using Fibonacci heaps. The algorithm is a type of greedy algorithm: it builds a set X vertex by vertex, always taking vertices closest to X.

[2] In this context, the terms "length" and "shortest" applied to a path refer to ℓ, not the number of edges in the path.

Algorithm 5.1 (Dijkstra's Algorithm)

$X := \{s\}$;
$D(s) := 0$;
for each $u \in V - \{s\}$ do
    $D(u) := \ell(s, u)$;
while $X \neq V$ do
    let $u \in V - X$ such that $D(u)$ is minimum;
    $X := X \cup \{u\}$;
    for each edge (u, v) with $v \in V - X$ do
        $D(v) := \min(D(v), D(u) + \ell(u, v))$
end while

The final value of $D(u)$ is $d(s, u)$. This algorithm can be proved correct by showing that the following two invariants are maintained by the while loop:

- for any u, $D(u)$ is the distance from s to u along a shortest path through only vertices in X;
- for any $u \in X$, $v \notin X$, $D(u) \leq D(v)$.

5.2 Reflexive Transitive Closure

Let E denote the adjacency matrix of the directed graph $G = (V, E)$. Using Boolean matrix multiplication, the matrix E^2 has a 1 in position uv iff there is a path of length exactly 2 from vertex u to vertex v; *i.e.*, iff there exists a vertex w such that $(u, w), (w, v) \in E$. Similarly, one can prove by induction on k that $(E^k)_{uv} = 1$ iff there is a path of length exactly k from u to v.

The reflexive transitive closure of G is

$$\begin{aligned} E^* &= I \vee E \vee E^2 \vee \cdots \\ &= I \vee E \vee E^2 \vee \cdots \vee E^{n-1} \\ &= (I \vee E)^{n-1} \ . \end{aligned}$$

The infinite join is equal to the finite one because if there is a path connecting u and v, then there is one of length at most $n - 1$.

Suppose that two $n \times n$ Boolean matrices can be multiplied in time $M(n)$. Then $E^* = (I \vee E)^{n-1}$ can be calculated in time $O(M(n) \log n)$ by squaring E $\log n$ times. We will show below how to calculate E^* in time $O(M(n))$. Conversely, if there is an algorithm to compute E^* in time $T(n)$, then $M(n)$ is $O(T(n))$ (under the reasonable assumption that $M(3n)$ is $O(M(n))$): to multiply A and B, place them strategically into a $3n \times 3n$ matrix, then take its reflexive transitive closure:

$$\begin{bmatrix} 0 & A & 0 \\ 0 & 0 & B \\ 0 & 0 & 0 \end{bmatrix}^* = \begin{bmatrix} I & A & AB \\ 0 & I & B \\ 0 & 0 & I \end{bmatrix} \ .$$

The product AB can be read off from the upper right-hand block.

Here is a divide and conquer algorithm to find E^* in time $M(n)$.

Algorithm 5.2 (Reflexive Transitive Closure)

1. Divide E into 4 submatrices A, B, C, D of size roughly $\frac{n}{2} \times \frac{n}{2}$ such that A and D are square.

$$E = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right]$$

2. Recursively compute D^*. Compute

$$F = A + BD^*C \ .$$

   Recursively compute F^*.

3. Set

$$E^* = \left[\begin{array}{c|c} F^* & F^*BD^* \\ \hline D^*CF^* & D^* + D^*CF^*BD^* \end{array}\right] \ .$$

Essentially, we are partitioning the set of vertices into two disjoint sets U and V, where A describes the edges from U to U, B describes edges from U to V, C describes edges from V to U, and D describes edges from V to V. We compute reflexive transitive closures on these sets recursively and use this information to describe the reflexive transitive closure of E. Note that we compute two reflexive transitive closures, a few matrix multiplications (whose complexity is given by M) and a few matrix additions (whose complexity is assumed to be quadratic) of matrices of roughly half the size of E. This gives the recurrence

$$T(n) = 2T(\frac{n}{2}) + cM(\frac{n}{2}) + d(\frac{n}{2})^2$$

where c and d are constants. Under the quite reasonable assumption that $M(2n) \geq 4M(n)$, the solution to this recurrence is $O(M(n))$.

5.3 All-Pairs Shortest Paths

Let E denote the adjacency matrix of a directed graph with edge weights. Replace the 1's in E by the edge weights and the 0's by ∞. Apply Algorithm 5.2 to calculate E^*, except use $+$ instead of $\wedge$ and min instead of $\vee$. We will show next time that this solves the all-pairs shortest path problem.