Course Overview

You ask computers to solve algorithmic problems on a daily basis. When you use google maps you ask Google to find the quickest route between a starting point and a destination point. When you use a search engine you ask it both to find the relevant pages from amongst the billions on the web and then to rank these pages so that the most interesting appear first. When you send an email, the web must decide how to route it to its destination. It must make this decision for millions of messages at once, and whilst so doing avoid overloading any of the communication channels between its servers.

It is imperative that the algorithms which solve such problems are efficient[1]. Whether this is the case depends crucially on the structures used to store the data. Fast algorithms require properly structured data. One important example of this fact comes from antiquity, and is taught in grade school. The arabic base 10 notation allows us to add two integers in a number of steps which is of the order of the sum of their logarithms and to multiply them in a number of steps which is quadratic in the sum of their logarithms. A simple every day example is retrieving the phone number of one of your contacts. If the names in your contact list appear in an arbitrary order then you may need to look at all the contacts before finding the one you need. If the list is an ordered array, then binary search allows you to find a contact in a number of steps which is logarithmic in the number of contacts.

This course presents technique for efficiently solving problems by considering appropriate data structures, and examples of problems that can be solved by doing so. Over half the course will be devoted to the graph data structure. We will discuss (i) how to formulate natural real world problems[2] as graph theory problems, and (ii) efficient algorithms for solving these graph theory problems. We note that reformulating a problem is essentially restructuring the way in which certain information is input while solving a problem typically involves manipulatng this structured information. We will also discuss: (i) sorting and selection, (ii) the priority queue data structure and its implementation via heaps, (iii) red-black trees, and (iv) compressing data so that it can be stored or transmitted efficiently (including the JPEG compression algorithm). For information on the required background and when we will cover specific topics, see the course web page.

[1] We consider an algorithm efficient if its worst case running time is polynomial in its input size

[2] including many that at first sight appear to have nothing to do with graph theory

Reformulations Using Boolean Formulas And Graphs

Because of the way they are built, traditionally computers have dealt with strings of zeros and ones, and to a large extent they still do so. Thus, reformulating a problem so that it can be solved via a computer usually involves turning its input into a string of 0s and 1s (or equivalently a false/true string). One way of doing so is to reformulate the problems so that its input involves BOOLEAN variables, that is variables which take the values TRUE and FALSE. For technical simplicity, in discussing such reformulations. we focus on *decision* problems, that is problems whose solution is either TRUE or FALSE.

One famous such problem is the SATISFIABILITY problem, which we define momentarily. As Cook showed, almost half a century ago, the famous P=NP? conjecture, for which there is a million dollar prize, is equivalent to the statement that there is an efficient algorithm to solve SATISFIABILITY. An easy corollary of Cook's result is that every instance I of a decision problem for which there is an efficient algorithm can be reformulated as an instance I' of SATISFIABILITY such that the size of I' is polynomial in the size of I. Indeed, as shown in class and below, these two statements remain true if we replace SATISFIABILITY by a special case 3-SAT in which the input boolean clauses have length three.

As we also showed in class and will show below, we can reformulate instances of the special case 2-SAT of SATISFIABILITY where clauses have length two as a graph theory problem. As we will show in the next lecture, this means we can solve it in linear time.

This suggests that our approach of reformulating problems as graph theory problems will have wide applicsbility, Further examples throughout the course will show that this is indeed the case.

Reducing SATISFIABILITY to 3-SAT

As mentioned earlier, a *boolean variable* x can be assigned the value true or the value false. Associated to x are two *literals* x and its negation $\neg x$. Assigning x the value true means x evaluates to true and $\neg x$ evaluates to false, while assigning x the value false means x evaluates to false and $\neg x$ evaluates to true. We note that the negation of $\neg x$ is x. I.e. $\neg\neg x = x$.

Boolean formulas over a set $X = \{x_1, ..., x_n\}$ of Boolean variables are obtained by combining the corresponding literals using AND, OR, and parentheses. A *truth assignment* for X is an assignment of true or false to each variable x_i (and hence to each literal). Using natural rules, this leads to an evaluation of every formula over the literals to either true of false. In particular, the OR of a set of subformulas evaluates to true precisely if any subformula evaluates to true while the AND of a set of subformulas evaluates to true precisely if all the subformulas evaluate to true.

SATISFIABILITY is concerned with the evaluation of a specially structured input formula. Specifically, a formula which is the AND of a set of subformulas called *clauses*. Each clause is the OR of a set of literals. Such an input formula is said to be in *conjunctive normal form*, or *CNF*.

A CNF formula evaluates to true with respect to a truth assignment to the variables precisely if in every clause there is a least one literal which evaluates to true. A truth assignment for which the formula evaluates to true is called *satisfying*.

The input for an instance of the SATIFIABILITY decision problem is a CNF formula. The question is whether there is a satisfying truth assignment. If so, the formula is called *satisfiable*. The size of a SATISFIABILITY instance is the sum over each clause of the number of literals in the clause.

For example (x_1 OR x_2 OR $\neg x_4$) AND (x_2 OR $\neg x_3$) AND x_5 has size 6 and is satisfiable; a satisfying truth assignment is obtained by assigning each x_i the value true. On the other hand, (x_1 OR $\neg x_2$) AND ($\neg x_1$ OR x_2) AND ($\neg x_1$ OR $\neg x_2$) AND (x_1 OR x_2) has size 8 and is not satisfiable.

For any positive integer k, the input to an instance of k-SAT is a CNF formula all of whose clauses have length k. We will be especially interested in 2-SAT and 3-SAT.

We now describe a linear time algorithm which given a CNF formula F returns a CNF formula F' all of whose clauses have length three, whose size is at most 3 times that of F, and such that F' is satisfiable precisely if F is. The existence of this reduction implies that if there is an efficient algorithm to solve 3-SAT then there is an efficient algorithm to solve SATISFIABILITY. The algorithm for SATISFIABILITY would simply first apply our reduction algorithm and then apply the algorithm for 3-SAT to its output.

So, suppose F has clauses C_1,...,C_k and is over the set $X = \{x_1, ..., x_n\}$ of Boolean variables. We let $l(i)$ be the length of C_i that is the number of literals C_i contains. Then letting $j_i = min\{1, l(i) - 2\}$ and $j = \sum_{i=1}^{k} j_i$, F' will have j clauses. For each i between 1 and l, there will be a set S_i of j_i of

these clauses which correspond to C_i.

We let $y_1^i, ..., y_{l(i)}^i$ be the literals occurring in C_i so
C_i is y_1^i OR y_2^i OR ... OR $y_{l(i)}^i$.
If $l(i) \leq 3$ then $j_i = 1$ and S_i is a single clause C_i'.
If C_i has length three then $C_i' = C_i$.
If C_i has length two then C_i' is y_1^i OR y_2^i OR y_2^i.
If C_i has length one then C_i' is y_1^i OR y_1^i OR y_1^i.
If $l(i) > 3$ then S_i has clauses $D_i^1, ...D_i^{l(i)-2}$ and uses new variables
$NOTSATISFIEDYET_2^i, ..., NOTSATISFIEDYET_{l(i)-1}^i$.
Clause D_i^1 is :
y_1^i OR y_2^i or $NOTSATISFIEDYET_2^i$.
For $2 \leq j \leq l(i) - 3$, Clause D_i^j is
$\neg NOTSATISFIEDYET_j^i$ OR y_{j+1}^i OR $NOTSATISFIEDYET_{j+1}^i$.
Clause $D_i^{l(i)-2}$ is
$\neg NOTSATISFIEDYET_{l(i)-2}^i$ OR $y_{l(i)-1}^i$ OR $y_{l(i)}^i$.

It is easy to see that the size of F' is at most 3 times the size of F. Furthermore, it is not difficult to construct the clauses of F' given those of F in linear time. It remains to show that F is satisfiable if and only if F' is satisfiable.

Now, if there is a satisfying truth assignment for F we can obtain one for F' as follows. We use the same assignment on X and for each i, j with $2 \leq j \leq l(i) - 2$ we set $NOTSATISFIEDYET_i^j$ to be true precisely if none of the literals $y_1^i, .., y_j^i$ evaluate to true.

On the other hand if there is a satisfying truth assignment for F' then its restriction to X is a satisfying truth assignment for F. To see this, it is enough to show that there is no satisfying truth assignment for F' such that for some i, all the y_j^i evaluate to false. Assume for a contradiction that such an assignment exists. Clearly $l(i)$ is at least 4. Considering clause D_i^1 we see that $NOTSATISFIEDYET_2^i$ must be assigned true. Then by induction on j we see that for $3 \leq j \leq l(i) - 2$, $NOTSATISFIEDYET_j^i$ is assigned true, by considering D_i^{j-1} and the inductive hypothesis. But now, clause $D_i^{l(i)-2}$ evaluates to false, a contradiction.

The (informal) description and proof of correctness of our reduction algorithm is complete.

Reducing 2-SAT to finding a good ordering

If for two Boolean literals x and y , we have that x OR y is a clause in an instance of 2-SAT then for any satisfying truth assignment $\neg x$ being true implies that y must be true and $\neg y$ being true implies that x must be true.

This motivates the definition of an *impliction directed graph* for an instance of 2-SAT. Its vertices are the literals corresponding to the set X of variables that the formula is over. Letting $\mathcal{C}$ be the set of clauses in the input formula, the arcs of the implication graph consist precisely of:

$$\{\neg x_i x_j | (x_i \ OR \ x_j) \in \mathcal{C}\} \cup \{x_i x_j | (\neg x_i \ OR \ x_j) \in \mathcal{C}\} \cup$$
$$\{\neg x_i \neg x_j | (x_i \ OR \ \neg x_j) \in \mathcal{C}\} \cup \{x_i \neg x_j | (\neg x_i \ OR \ \neg x_j) \in \mathcal{C}\}.$$

We note that we think of the clauses as unordered so each clause gives rise to two arcs. It is easy to see that a truth assignment is satisfying for the formula precisely if there is no arc of the implication graph from a literal which evaluates to true to a literal which evaluates to false. It is also easy that we can construct this implication graph from the clauses of an instance of 2-SAT in linear time. Thus we can reduce a 2-SAT instance to an instance of the following problem in linear time:

Definition: we say a labelling of the vertex set of an implication graph with T and F is *good* if (i) for every i, x_i and $\neg x_i$ get different labels and (ii) there is no arc yz such that y is labelled T and z is labelled F.

Problem: IS THERE A GOOD LABELLING?

Input: Directed implication graph G with vertex set $V = \{x_1, ..., x_n, \neg x_1, ..., \neg x_n\}$ and arc set A.

Question: Is there a good labelling of $V(G)$?

We note that if the desired labelling exists and xyz is a path such that x is labelled T then by considering the edge xy so is y. Hence by considering the edge yz, so is z. More generally we see that our condition on the desired labelling is equivalent to the statement that if x is labelled T then any vertex which is on a directed path starting at x is also labelled T.

We recall that a strong component of a digraph is a maximal set of vertices such that for any two vertices u and v in the set, there is a path from u to v. We note that if there is a path from u to v and a path from v to w then there is a path from u to w. Hence the vertex sets of strong components are disjoint and partition $V(G)$.

We note that our desired condition implies that if any vertex of a strong component of G is labelled T all of its vertices must be labelled T. So we have:

Lemma: If an implication graph has a good labelling then none of its strong components contain both x_i and $\neg x_i$ for any i.

Proof One of x_i or $\neg x_i$ must be labelled T and the other F. □

It turns out, as we now explain, that the desired labelling exists precisely if there is no i such that x_i and $\neg x_i$ lie in the same strong component. More strongly, we show now that there is a linear time algorithm which when applied to an implication graph, finds a labelling with T and F which has the desired property provided no such i exists.

Key to doing so is constructing a special ordering on the vertices of a graph. We will be interested in orderings where the vertices of each strong component appear consecutively in the ordering. Furthermore, we will insist that any edge between strong components must go from some vertex to a vertex which appears later in the order.

Definition: We say that an order on the vertices of a directed graph G is *good* if we can label the strong components of G as $Z_1, ..Z_k$ so that for $i < j$ all the vertices of Z_i appear before all the vertices of Z_j under the ordering, and for all $i < j$ there is no arc from a vertex of Z_j to a vertex of Z_i.

In the next class, we shall present and discuss an algorithm with the following specifications.

Algorithm: FINDING A GOOD ORDER

Input: A directed graph G.
Output: A linked list ORD which lists the vertices of G in a good order.
Running time: $O(|V(G)| + |E(G)|)$.

We then exploit the following result:

Lemma: Suppose G is a directed graph in which no literal is contained in the same strong component as its negation. Then for any good ordering of G, labelling x_i with T if it appears after $\neg x_i$ in the order and F otherwise yields a good labelling.

Proof: We note that by the construction of our implication graph, if yz is an arc so is $\neg y \neg z$. It follows by induction on the length of the path that if there is a directed path from y to z there is a path from $\neg y$ *to* $\neg z$. In the

same vein if there is a path from z to y then there is a path from $\neg y$ to $\neg z$. Thus, if for every literal x, we let S_x be the strong component containing x, then we have that $S_{\neg x} = \{\neg y | y \in S_x\}$. Hence the vertex sets of the strongly connected components split into a set of pairs each of which is S_x and $S_{\neg x}$ for some x and is also S_y and $S_{\neg y}$ for every y in $S_x \cup S_{\neg x}$.

Now, by the definition of a good order, since x and $\neg x$ lie in different strong components either all the vertices of S_x come before all those of $S_{\neg x}$ or all the vertices of $S_{\neg x}$ come before all those of S_x. It follows that for every strong component, our labelling procedure either labels all its vertices T or all its vertices F. Thus if there is an arc xy such that x has the label T and y has the label F then x and y lie in different strong components.

Because of the pairing of the strong components, so do $\neg x$ and $\neg y$. Now, because xy is an arc, the vertices of S_x appears before those of S_y in the order. Because of the way that our implication graph was constructed, $\neg y \neg x$ is an arc of this graph. Hence the vetices of $S_{\neg y}$ appears before those of $S_{\neg x}$ in the order. Because x is labelled true the vertices of $S_{\neg x}$ appears before the vertices of S_x in the order. Hence, the vertices of $S_{\neg y}$ appears before those of S_y and y is labelled true, a contradiction. □

We can now set out our linear time algorithm for solving an instance G of IS THERE A GOOD LABELLING? It uses an array LABEL indexed by the vertex set V of G.The entries of LABEL can be T,F, or $\emptyset$. It returns either a good labelling in LABEL or the information that no such labelling exists

(1) apply FINDING A GOOD ORDER to G.

(2) For all v in V do $LABEL[v] = \emptyset$

(3) Traverse ORD; when we come to v perform the following:
IF $LABEL[v] = \emptyset$ THEN $LABEL[v] := F$, $LABEL[\neg v] := T$ ENDIF

(4) For each uv in $E(G)$ IF $LABEL[u] = T$ AND $LABEL[v] = F$ THEN return "no good labelling exists" and terminate ENDIF

(5) Return LABEL and terminate.