Lecture Note for Graph Algorithms

Gerard Jennhwa Chang
Department of Mathematics, National Taiwan University

http://www.math.ntu.edu.tw/∼gjchang/

February 22, 2005

1 Terminology in graph theory

A *graph* $G = (V, E)$ is a structure which consists of a finite nonempty set V of *vertices* and a set E of *edges* that are unordered pairs of distinct vertices. We very often use uv as a short notation for $\{u, v\}$. Vertices u and v are called the *end-vertices* of an edge uv. Two vertices are *adjacent* if they form an edge; two edge are *adjacent* if they have a common end-vertex; an edge is *incident* to its end-vertices. For visual convenience, we often draw a graph by placing each vertex at a point and representing each edge by a curve joining the locations of its end-vertices, see Figure 1 for an example.

Figure 1: The graph $P_4 = (V, E)$ with $V = \{v_1, v_2, v_3, v_4\}$ and $E = \{v_1v_2, v_2v_3, v_3v_4\}$.

We remark that sometimes we allow *parallel* (or *multiple*) *edges* between two vertices; in this case we have *multi-graphs*. We may also allow an"edge," which is called a *loop*, having two identical end-vertices; in this case we have *pseudo-graphs*. By considering the "edges" as ordered pairs rather than unordered pairs, we have *digraphs*, *multi-digraphs* and *pseudo-digraphs*.

For two vertices x and y, an *x-y walk* (or a *walk from x to y*) is a sequence

$$W : x_0, e_1, x_1, e_2, \ldots, x_{n-1}, e_n, x_n,$$

where $x_0 = x$, $x_n = y$ and e_i is an edge with end-vertices x_{i-1} and x_i for $1 \leq i \leq n$. The *length* of the walk is n, which can be 0. Notice that we may simply use $x_0, x_1, \ldots, x_n$ for a walk as each e_i is determined by x_{i-1} and x_i. We write the definition in this way just for the purpose that it is also useful for multi-graphs. A *tour* is a walk in which no edge is repeated.

A *path* is a walk in which no vertex is repeated. A *cycle* is a walk in which no vertex is repeated except $x_0 = x_n$.

A graph $G' = (V', E')$ is a *subgraph* of another graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. For a nonempty subset S of V, the subgraph of $G = (V, E)$ *induced* by S is the graph $G[S] = (S, E[S])$, where $E[S] = \{uv \in E : u \in S, v \in S\}$.

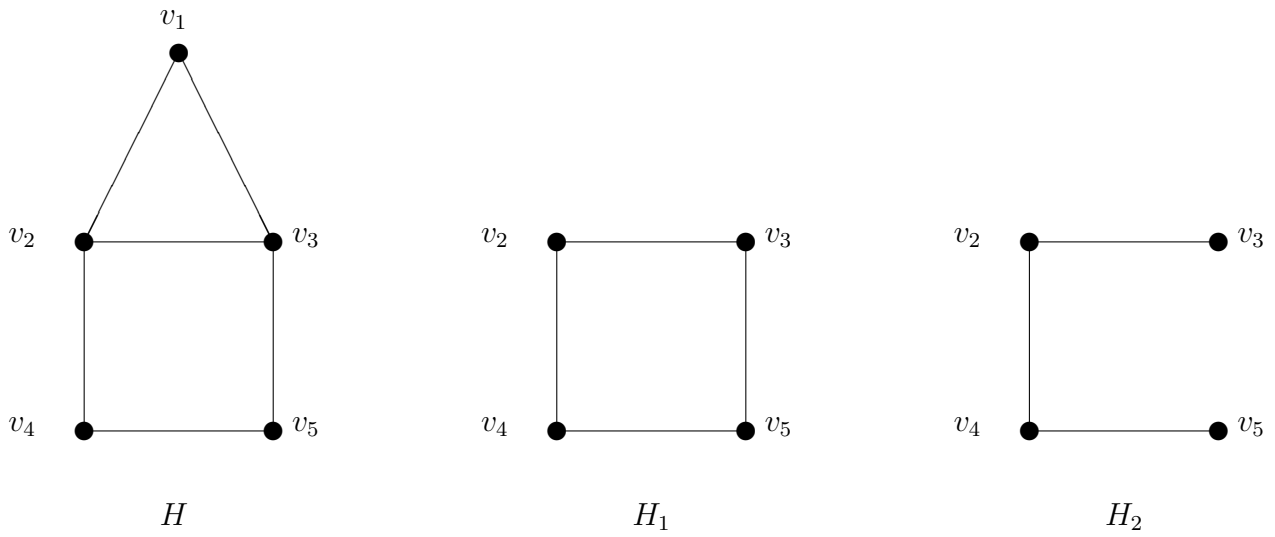


Figure 2: The house graph H and two of its subgraphs H_1 and H_2.

We close this section by using the house graph H in Figure 2 to demon the definitions given above. First, v_1, v_2, v_3, v_1, v_2 is a walk but is not a tour; v_2, v_3, v_1, v_2, v_4 is a tour but is not a path; v_1, v_2, v_4, v_5 is a path; and v_2, v_3, v_5, v_4, v_2 is a cycle. While H_1 is an induced subgraph of H, the subgraph H_2 is not induced.

2 A starting example—graph connectivity

Traditionally, mathematics results are presented in the form of proposition. In this computer age, algorithmic aspects of mathematics are more and more popular. In this section, we shall use graph connectivity to show the difference between these two approaches, while similarity hinds inside.

A graph is *connected* is for every two vertices x and y, there is an x-y walk. In the definition, we in fact can replace "walk" by "path" due to the following result. A graph is *disconnected* if it is not connected.

Homework 1 *For any two vertices x and y in a graph, there is an x-y walk if and only if there is an x-y path.*

Having the definition, a basic question is how to determine if a graph is connected. If the graph is described formally, we may apply the definition directly, see the examples below.

Example. The *n-cube* is the graph $Q_n = (V, E)$ with $V = \{x = x_1x_2 \dots x_n : x_i \in \{0, 1\}$ for $1 \le i \le n\}$ and $E = \{xy : \sum_{i=1}^{n} |x_i - y_i| = 1\}$. Notice that in the n-cube, two vertices are adjacent if and only if they differ at exactly one bit position. We claim that the n-cube is connected as follows. For any two vertices x and y, let $j_1, j_2, \dots, j_r$ are all the indices j for which $x_j \neq y_j$. Then $x^{(0)}, x^{(1)}, \dots, x^{(r)}$ is an x-y path, where $x^{(0)} = x$ and $x^{(i)}$ is obtained from $x^{(i-1)}$ by changing j_ith bit from $x_{j_i}^{(i-1)}$ to $1 - x_{j_i}^{(i-1)}$ for $1 \le i \le r$. □

Homework 2 *For integers $n \ge k \ge 1$, the Kneser graph is the graph $\mathrm{KG}(n, k) = (V, E)$ with $V = \{A : A \subseteq \{1, 2, \dots, n\}, |A| = k\}$ and $E = \{AB : A \in V, B \in V, A \cap B = \emptyset\}$. Prove that if $n \ge 2k + 1$, then $\mathrm{KG}(n, k)$ is connected.*

We remark that $\mathrm{KG}(5, 2)$ is the Petersen graph.

In general, we may like to treat the connectivity for general graphs. We now see a typical proposition on connectivity as follows. The *complement* of a graph $G = (V, E)$ is the graph $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{uv : u \in V, v \in V, uv \notin E, u \neq v\}$.

Proposition 1 *If $G = (V, E)$ is disconnected, then $\overline{G}$ is connected.*

Proof. As G is disconnected, there are two vertices x and y such that there is no x-y walk. Let A (respectively, B) be the set of vertices z such that there is an (respectively, there is no) x-z walk in G. Notice that $x \in A$ and $y \in B$; and so V is partitioned into two nonempty sets A and B.

Suppose there is an edge $uv \in E$ with $u \in A$ and $v \in B$. As there is an x-u walk in G and $uv \in E$, there is an x-v walk in G. Hence, v must be in A, which contradicts to the fact that $v \in B$. Consequently, for any vertices $u \in A$ and $v \in B$ we have $uv \notin E$ or $uv \in \overline{E}$.

To prove the connectivity of $\overline{G}$, we consider two cases for any two vertices u and v in V. For the case when $u \in A$ and $v \in B$, or $u \in B$ and $v \in A$, we have the u-v walk u, v in $\overline{G}$ as $uv \in \overline{E}$. For the case when $u, v \in A$ (respectively, $u, v \in B$), choose a vertex z in B (respectively, A) and then u, z, v is a u-v walk in $\overline{G}$. Thus, $\overline{G}$ is connected. □

The converse of this proposition is not always true as shown by the example P_4 in Figure 1. A graph is *G-free* if it has no G as an induced subgraph. The converse of Proposition 1 holds when the graph is P_4-free.

Homework 3 *If G is connected and P_4-free, then $\overline{G}$ is disconnected.*

It is an interesting question in the algorithmic aspect of graph theory that how to test if a graph is P_4-free. We will be back to this question later.

February 24, 2005

There are many other propositions on graph connectivity. Most of them may not be so useful to answer the question if a graph is connected. The algorithmic aspect of this problem arises now. Suppose $G = (V, E)$ is a graph for which we want to know if it is connected. Here is a possible way to do this.

Algorithm CONN

```
1.    choose a vertex x from V and let Q = {x} ;
2.    mark x and let all other vertices unmarked ;
3.    while (Q ≠ ∅)
4.    {   get a vertex v out of Q ;
5.        for all vertices u adjacent to v do
6.            if u is unmarked then mark u and put it into Q ;
7.    }
8.    if all vertices are marked then output "connected" else output "disconnected".
```

We next consider how to implement the algorithm.

3 Implementation and data structure

We only assume the basic data structure array. In mathematics, we often consider sequences such as $x_1, x_2, \ldots, x_n$. In computer, we use x[1], x[2], ..., x[n] to represent the sequence, and often use x[1..n] as a short notation for it. We use M[1..m][1..n] to represent a two dimensional array of size $m \times n$, which is also know as an $m \times n$ matrix.

To implement an algorithm like CONN, we first need a way to represent a graph $G = (V, E)$. For convenience, we often assume that $V = \{1, 2, \ldots, n\}$. From the definition, we in fact have an $m \times 2$ matrix E[1..m][1..2] in which E[i][1] and E[i][2] are the end-vertices of the ith edge. As an example, to represent P_4, we have $n = 4$, $m = 3$, E[1][1]=1, E[1][2]=2, E[2][1]=2, E[2][2]=3, E[3][1]=3 and E[3][2]=4.

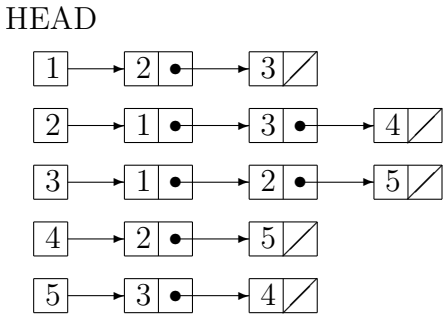
This representation for a graph is often quit inconvenient. For instance, when we want to execute line 5 of CONN, we need to scan through all edges of the graph even most of them are not related to vertex v.

One simple minded way to resolve the problem is to create the *adjacency matrix* of the graph, which is the $n \times n$ matrix A[1..n][1..n] in which A[i][j] = 1 if i is adjacent to j and A[i][j] = 0 otherwise. The matrix A is zero-one and symmetric with diagonal all zero. The following is the adjacency matrix for the house graph H in Figure 2.

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} \begin{array}{c} \begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & v_5 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

Now, to execute line 5 of CONN, we only need to scan the vth row of the adjacent matrix. Those u with A$[v][u] = 1$ are vertices adjacent to v. The drawback of this method happens when the graph is sparse, namely, when there are much zero than one in the matrix or $m << n(n-1)/2$.

An alternative way to store the graph is the *adjacency list structure*. This method uses a link list for each vertex. The list for a vertex contains all vertices adjacent to it. We need three arrays to represent the structure: HEAD$[1..n]$, DATA$[1..2m]$ and NEXT$[1..2m]$.



	1	2	3	4	5
HEAD	1	2	4	8	10

	1	2	3	4	5	6	7	8	9	10	11	12
DATA	2	1	3	1	3	2	4	2	5	3	5	4
NEXT	3	5	0	6	7	9	0	11	0	12	0	0

Figure 3: The adjacency list of the house graph H in Figure 2

March 1, 2005

To implement CONN, we need arrays Q$[1..n]$, MARK$[1..n]$, HEAD$[1..n]$, DATA$[1..2m]$, NEXT$[1..2m]$. We then have the following detailed program.

Algorithm CONN-implementation

call input-graph ;
1. QBEG = QEND = 1 ; Q[QEND] = 1 ;
2. MARK[1] = 1 ; **for** $i = 2$ **to** n **do** MARK$[i] = 0$;
   $nn = 1$;
3. **while** (QBEG $\leq$ QEND)
4. { v = Q[QBEG] ; QBEG = QBEG + 1 ;
   p = HEAD$[v]$;
5. **while** $(p \neq 0)$
   { u = DATA$[p]$; p = NEXT$[p]$;
6. **if** MARK$[u] = 0$
   then { MARK$[u] = 1$; QEND = QEND + 1 ; Q[QEND] = u ; $nn = nn + 1$ };
   }
7. }
8. **if** $nn = n$ **then output** "connected" **else output** "disconnected".

The running time of this algorithm is $O(|V| + |E|)$.

Although not necessary, the set Q is implemented in the way of first-in-first-out. This kind of data structure is called a *queue*. Sometimes a set may be implemented as last-in-first-out. In this case we have a *stack*.

The method for the graph connectivity is also known as the *breadth-first search*, which is a frequently used method in the design of algorithms. It can also be used to find all connected components, to check if a graph is a tree, to check if a graph is bipartite, to check if a graph is chordal ... etc. We close this section by assigning some of them as homework. First, some definitions.

A (*connected*) *component* of a graph is a maximal subgraph of the graph. Consequently, a component must be an induced subgraph. Suppose $G = (V, E)$ has r components $G_i = (V_i, E_i)$ for $1 \leq i \leq r$. It is the case that V is partitioned into $V_1, V_2, \ldots, V_r$; and E is partitioned into $E_1, E_2, \ldots, E_r$. By finding the components of G, it is only necessary to find all V_is.

Homework 4 *Design an algorithm for finding all components of a graph $G = (V, E)$.*

A *tree* is a connected graph without cycle.

Homework 5 *Design an algorithm to test if a graph $G = (V, E)$ is a tree.*

A graph $G = (V, E)$ is *bipartite* is V can be partitioned into two nonempty sets A and B such that each edge has one end-vertex in A and the other in B.

Homework 6 *Design an algorithm to test if a graph $G = (V, E)$ is bipartite.*

4 Eulerian tours

In 1736, Euler published a paper solving the seven bridge problem. This started the field of graph theory.

An *Eulerian tour* of a graph is a tour which uses all edges of the graph and whose two end-vertices are identical. Not all graphs have Eulerian tours. For instance, the house graph H in Figure 2 is not Eulerian, while H_1 is.

To see the answer from Euler, we need a definition. In a graph G, the *degree* of a vertex x, denoted by $\deg_G(x)$ or $\deg(x)$, is the number of edges incident to x. This definition also works for multi-graphs. If we count a loop to be incident to its end-vertex twice, this definition also works for pseudo-graphs.

Lemma 2 *In a pseudo-graph $G = (V, E)$, $\sum_{x \in V} \deg(x) = 2|E|$.*

Proof. When counting the degrees of all vertices, every edge is counted twice. □

Corollary 3 *In a pseudo-graph, there is an even number of vertices of odd degree.*

Here comes the main theorem from Euler.

Theorem 4 *A pseudo-graph G without isolated vertex is Eulerian if and only it is connected and every vertex is of even degree.*

Proof. The necessity is clear. For the sufficiency, choose a longest tour in G:

$$W : x_0, e_1, x_1, e_2, \ldots, x_{n-1}, e_n, x_n.$$

First, $x_0 = x_n$ for otherwise there is only an odd number of edges incident to x_n appear in W. As the degree of each vertex is even, we then can find an edge e_{n+1} with end-vertices x_n and x_{n+1}, which is not in W. Adding e_{n+1}, x_{n+1} at the end of W results a longer tour, a contradiction to the choice of W.

Suppose W does not contain all edges of G. As G is connected, there is a vertex x_i in W incident to an edge e not in W. Let x be the other end-vertex of e. Then

$$x, e, x_i, e_{i+1}, x_{i+1}, e_{i+2}, \ldots, x_{n-1}, e_n, x_n = x_0, e_1, x_1, e_2, x_2, \ldots, e_i, x_i$$

is a tour longer than W, again a contradiction.

Therefore, W is an Eulerian tour. □

We may turn the proof of the theorem above to an algorithm by keeping a walk W as a doubly linked list. Initially, W contains only one vertex x which is chosen arbitrarily. we also have a pointer v which is x at the first. Initially, all edges are unmarked. At any iteration, if G has a unmarked edge vu, mark this edge vu, insert the vertex u after v, and use u as the next pointer v. On the other hand, if all edges incident to v are marked, then replace the pointer v by the vertex previous to v in the doubly linked list W and repeat the same procedure. The program terminates when v is the first vertex of the list W.

To implement the program properly, in the adjacent list structure for G, we not only need HEAD[1..n], DATA[1..2m], NEXT[1..2m] but also CURRENT[1..n], MARK[1..2m], MATE[1..2m]. The purpose of CURRENT[i] is to point to the current un-scanned edge incident to vertex i. Note that DATA[j] not only representing a vertex adjacent to some vertex i, it also corresponds an edge incident to i. The algorithm also works for parallel edges when the list contains a same vertex appears more than once, and for a loop when DATA[j] is i. The purpose of MARK[j] is to record if the edge corresponding to DATA[j] is marked or not. The purpose of MATE[j] is the point to the other end vertex for the vertex corresponding to DATA[j]. When we mark an edge DATA[j] in the list of vertex i, we also need to mark its mate in the other list.

Using the data structure we may implement the algorithm in $O(|V| + |E|)$ time.

March 3, 2005

A pseudo-digraph is *strongly connected* if for any two vertices x and y there is an x-y di-walk.

The above algorithm also works for di-pseudo-graphs. A couple of revisions are as follows.

First, the *out-degree* $\deg^+(v)$ of a vertex v is the number of edges from v to all vertices in G; the *in-degree* $\deg^-(v)$ of a vertex v is the number of edges from all vertices in G to v.

Similar to the graph case, we also have the following results. The proofs are similar and so are omitted.

Lemma 5 *In a pseudo-digraph $G = (V, E)$, $\sum_{x \in V} \deg^+(x) = \sum_{x \in V} \deg^-(x) = |E|$.*

Theorem 6 *A pseudo-digraph G without isolated vertex is Eulerian if and only it is strongly connected and out-degree equals in-degree for any vertex.*

The implementation is also the same for graphs, except we don't need the array MATE[1..2m] now.

An interesting application of the theorem above is the existence of de Bruijn sequences. Let $\Sigma = \{1, 2, \ldots, \sigma\}$ is an alphabet of σ letters. Clearly, there are σ^n different words of length n over Σ. A *de Bruijn sequence* is a (circular) sequence $a_0 a_1 \ldots a_{L-1}$ over Σ such that every word w of length n over Σ there exists a unique i such that

$$a_i a_{i+1} \ldots a_{i+n-1} = w,$$

where the computation of the indices is modulo L. Clearly if the sequence satisfies this condition, then $L = \sigma^n$. The most important case is that $\sigma = 2$. Binary de Bruijn sequences are of great important in the coding theory and are implemented by shift registers.

Let us describe a pseudo-digraph $G_{\sigma,n} = (V, E)$, where V is the set of all σ^{n-1} words of length $n-1$ over Σ; E is the set of all σ^n words $b_1 b_2 \ldots b_n$ of length n over Σ, which is view as an edge from the vertex $b_1 b_2 \ldots b_{n-1}$ to the vertex $b_2 b_3 \ldots b_n$.

Theorem 7 *For every positive integers σ and n, $G_{\sigma,n}$ has a directed Eulerian tour, which corresponds to a de Bruijn sequence.*

Proof. First, every vertex is out out-degree σ and in-degree σ. For any vertices $b = b_1 b_2 \ldots b_{n-1}$ and $c = c_1 c_2 \ldots c_{n-1}$, there is a b-c walk as follows:

$$b_1 b_2 \ldots b_{n-1}, b_2 b_3 \ldots b_{n-1} c_1, b_3 b_4 \ldots b_{n-1} c_1 c_2, \ldots, b_{n-1} c_1 c_2 \ldots c_{n-2}, c_1 c_2 \ldots c_{n-1}.$$

This proves that $G_{\sigma,n}$ is strongly connected. According to Theorem 6, $G_{\sigma,n}$ has an Eulerian tour. Picking the first bit corresponding to an edge in the order of the tour, we then have a de Bruijn sequence as desired. □

Notice that the pseudo-digraph $G_{\sigma,n}$ has σ^{n-1} vertices and σ^n edges. These two numbers are far greater than σ and n. We conclude this section with the following interesting homework.

Homework 7 *Find a de Bruijn sequence for α and n without explicitly constructing $G_{\sigma,n}$.*

There are many papers of this line can be found from the database like SCI.

March 8, 2005

5 Shortest path problem

In a graph G the *distance* from a vertex u to another vertex v, denoted by $d_G(u,v)$ or $d(u,v)$, is the minimum length of an u-v walk; the minimum is ∞ if there is no such walk.

For a fixed vertex x the breadth-first search gives the distances from x to all vertices y. This can be done by modifying the Algorithm CONN by setting $d(x) = 0$ and $d(y) = \infty$ for all $y \neq x$ initially, and in line 6 let $d(u) = d(v) + 1$. An induction, see Theorem 9 below, can be used to prove that at the termination of the algorithm $d(v) = d(x,v)$ for all vertices v.

Suppose each edge e of graph G has a weight $w(e)$ which is a positive real number. The weighted distance from u to v, denoted by $d_w(u,v)$ is the minimum $\sum_{e\in W} w(e)$ where W runs over all u-v walk. Dijkstra gave the following algorithm for finding the weighted distance from a fixed vertex x to all other vertices.

Assume $w(u,v)$ is given all vertices u and v, where $w(u,u) = 0$ and $w(u,v) = \infty$ if there is no edge uv.

Algorithm DIJKSTRA
1. let $d(y) = w(x,y)$ for all vertices y ;
2. let $Q = \{x\}$; $p(x) = 0$; $p(y) = x$ for all $y \neq x$;
3. **while** $(Q \neq V)$
4. { choose a vertex v not in Q with minimum $d(v)$;
5. $Q = Q \cup \{v\}$;
6. **for all** vertices u not in Q **do**
7. **if** $d(v) + w(v,u) < d(u)$ **then** $\{d(u) = d(v) + w(v,u); p(u) = v;\}$;
8. }

The algorithm can be implemented in $O(|V|^2)$ time. We now prove the correctness of the algorithm. In fact, we add also the proofs of correctness of other algorithms.

Theorem 8 *Algorithm CONN correctly determines if a graph is connected.*

Proof. We first claim that a vertex v is marked if and only if there is an x-v walk.

Suppose v_i is the ith vertex marked in the algorithm. We shall prove by induction on i that there is an x-v_i walk W_i. For the case of $i = 0$, we have $v_0 = x$ and so we may choose

$W_0 = x$. Suppose $i \geq 1$ and there is an x-$v_{i'}$ walk $W_{i'}$ for all $i' < i$. By lines 4 and 5 of the algorithm, there is some $j < i$ such that $v_j v_i$ is an edge. Then, by the induction hypothesis, there is an x-v_j walk W_j. And hence $W_i = W_j + v_j v_i$ is an x-v_i walk. These prove the necessity of the claim.

On the other hand, suppose there is an x-v walk $x = v_0, v_1, \ldots, v_r = v$ but v is unmarked. As v_0 is marked, there is some v_j marked but v_{j+1} unmarked. When the algorithm runs lines 4 and 5 at the iteration of $v = v_j$ and $u = v_{j+1}$,the vertex v_{j+1} must be marked, which is a contradiction. So the claim holds.

To see the correctness of the algorithm. Suppose all vertices are marked. For any two vertices v_i and v_j we have an v_i-v_j walk $W_i +$ reverse(W_j). This shows that G is connected. On the other hand, suppose there is some vertex v unmarked. By the claim above, there is no x-v walk and so G is disconnected. □

Theorem 9 *The revised CONN correctly determines the distance $d(x, v)$, namely, $d(v) = d(x, v)$ for all vertices v of the graph.*

Proof. By the claim in the proof of Theorem 8, $d(v) < \infty$ if and only if $d(x, v) < \infty$. So the theorem is true for the case when $d(v) = \infty$ or $d(x, v) = \infty$. We then may assume that $d(v)$ and $d(x, v)$ are finite.

Let $p(u) = v$ in line 5 of the algorithm. It is then the case that $d(u) = d(p(u)) + 1$ and $p(u)$ is the first marked neighbor of u.

We first prove that $d(v) \geq d(x, v)$ by induction on $d(v)$. For the case when $d(v) = 0$, we have $v = x$ and so $d(v) = 0 = d(x, v)$. Suppose $d(v) \geq 1$ and $d(v') \geq d(x, v')$ for v' with $d(v') = d(v) - 1$. As $d(p(v)) = d(v) - 1$, by the induction hypothesis, $d(p(v)) \geq d(x, p(v))$. Then $d(v) = d(p(v)) + 1 \geq d(x, p(v)) + 1 \geq d(x, v)$. Notice that the last inequality follows from that an x-$p(v)$ walk of length $d(x, p(v))$ together with the edge $p(v)v$ form an x-v walk of length $d(x, p(v)) + 1$.

To prove $d(v) \leq d(x, v)$, we first prove the following claim.

Claim. If u is marked before v then $d(u) \leq d(v)$.

We shall prove the claim by induction on $d(u)$. The case when $d(u) = 0$ is clear as $u = x$. Suppose $d(u) \geq 1$ and the claim holds for u' with $d(u') = d(u) - 1$. Notice that $d(p(u)) = d(u) - 1$ and $d(p(v)) = d(v) - 1$. As u is marked before v, according to the algorithm, it is also the case that $p(u)$ is marked before $p(v)$. By the induction hypothesis, $d(p(u)) \leq d(p(v))$ and so $d(u) \leq d(v)$.

Having this claim at hand, we next prove that $d(v) \leq d(x, v)$ by induction on $d(x, v)$. For the case when $d(x, v) = 0$, we have $v = x$ and so $d(v) = 0 = d(x, v)$. Suppose $d(x, v) \geq 1$ and $d(v') \leq d(x, v')$ for v' with $d(x, v') = d(x, v) - 1$. Let v' be the vertex before v in a shortest x-v path. As $d(x, v') = d(x, v) - 1$, by the induction hypothesis, $d(v') \leq d(x, v')$. As $p(v)$ is the first marked neighbor of v, by the claim above, $d(p(v)) \leq d(v')$ and so $d(v) = d(p(v)) + 1 \leq d(v') + 1 \leq d(x, v') + 1 = d(x, v)$. Hence the theorem holds. □

March 10, 2005

Theorem 10 *Algorithm DIJKSTRA correctly determines the weighted distance $d_w(x, v)$, namely $d(v) = d_w(x, v)$, for all vertices v of the graph.*

Proof. We first observe the following facts.
(a) $d(v) \geq 0$ at any iteration.
(b) $d(v)$ decreases from time to time, and remains unchanged after it is put into Q.
(c) $v, p(v), p(p(v)), p(p(p(v))), \ldots, x$ is the reverse of an x-v walk with weighted length $d(v)$.
(d) $d(v) + w(v, u) \geq d(u)$ for all $v \in Q$ and $u \notin Q$.

We shall prove that at any iteration, $d(v) = d_w(x, w)$ for all $v \in Q$.

First, $d(v) \geq d_w(x, v)$ by fact (c). We shall prove $d(v) \leq d_w(x, v)$ by induction on $|Q|$. According to (b), we in fact only need to prove this for the vertex added into Q in line 2 or 5. The claim holds for $|Q| = 1$ as x is put into Q by line 2. Suppose $|Q| \geq 1$ and v is put into Q to get $Q' = Q \cup \{v\}$. Choose a shortest x-v path $W : x, \ldots, v', v'', \ldots, v$, where v'' is the first vertex of W not in the set Q. Then,

$$\begin{array}{rcll} d_w(x, v) & = & \text{length}_w(W) & \\ & \geq & \text{length}_w(W(x, v')) + w(v', v'') & \text{(weights of edges are positive)} \\ & \geq & d_w(x, v') + w(v', v'') & (W(x, v') \text{ is an } x\text{-}v' \text{ path}) \\ & = & d(v') + w(v', v'') & \text{(by the induction hypothesis)} \\ & \geq & d(v'') & \text{(by fact (d))} \\ & \geq & d(v) & \text{(by the choice of } v \text{ in line 4.)} \end{array}$$

The theorem then follows. □

We close this section by remark that the above arguments are also valid for digraphs. There are also algorithms for digraphs having some edges with negative weights.

6 Minimun spanning trees

In the following few sections, we shall consider algorithms on trees. We first make an introduction to trees. Recall that *trees* are connected graphs without cycles. A *leaf* is a vertex of degree one. A *spanning subgraph* of a graph $G = (V, E)$ is a graph $G' = (V', E')$ with $V' = V$ and $E' \subseteq E$. A *spanning tree* is a spanning subgraph that is a tree.

Lemma 11 *If xy is an edge in a cycle C of a connected graph G, then $G - xy$ is still connected.*

Proof. Suppose u and v are any two vertices in $G - xy$. Choose a u-v walk in G. If we replace any occurrence of the edge xy in the walk by $C - xy$, we get a u-v walk in $G - xy$. This gives that $G - xy$ is connected. □

Corollary 12 *A graph G has a spanning tree T if and only if G is connected.*

Proof. The sufficiency follows from that a u-v walk in T is also a u-v walk in G. On the other hand, suppose G is connected. Choose a connected spanning subgraph T of G with as few edges as possible. Suppose T has a cycle C. Choose an edge xy from the cycle. By Lemma 11, $T - xy$ is a connected spanning subgraph of G with fewer edges than T, which is a contradiction. Thus, T is a spanning tree of G. □

Lemma 13 *Any tree T of at least two vertices has at least two leaves x and y. And, $T - x$ is a tree with one vertex and one edge fewer than T.*

Proof. Choose a longest path $P : x, x', \ldots, y', y$ in T. As T has at least one edge, $x \neq y$. Suppose x is not a leaf. Then x has another neighbor $x'' \neq x'$. If $x'' \in P$, then T has a cycle, which is impossible. Now, $x'' \notin P$ gives a path x'', P longer than P, a contradiction. This proves that x is a leaf. Similarly, y is a leaf.

Suppose z is the only neighbor of x in T. Then, $T - x$ is fewer than T by vertex x and edge xz. For any two vertex u and v in $T - x$, choose a u-v path P in T. As u and u are not in T, the vertex x is not in P for otherwise it has degree at least two. Hence, P is also a u-v path in $T - x$. This proves that $T - x$ is connected. □ .

Corollary 14 $|V| = |E| + 1$ *for any tree $T = (V, E)$.*

Proof. The corollary is clear for $|V| = 1$. Suppose $|V| \geq 2$ and $|V'| = |E'| + 1$ for any tree $T' = (V', E')$ with $|V'| = |V| - 1$. By Lemma 13, we may choose a leaf x in T. Then $T' = T - x$ is a tree with $|V'| = |V| - 1$ and $|E'| = |E| - 1$. By the induction hypothesis, $|V'| = |E'| + 1$ and so $|V| = |V'| + 1 = (|E'| + 1) + 1 = |E| + 1$. □

The following are some equivalent definitions of trees.

Theorem 15 *The following statements are true for any graph $T = (V, E)$.*
(a) *T is a tree.*
(b) *T is a connected graph such that deleting any edge from T results a disconnected graph.*
(c) *T is a graph without cycle such that adding an edge e not in T creates a cycle.*
(d) *There is a unique path u-v path between any two vertices u and v in T.*
(e) *T is connected and $|V| = |E| + 1$.*
(f) *T has no cycle and $|V| = |E| + 1$.*

March 15, 2005

We remark that the cycle created in (c) must contain the edge e and is unique, which is called the *fundamental cycle* of T with respect to the edge e and is denoted by $C(T, e)$. Also deleting any edge e' in $C(T, e)$ from $T + e$ results again a tree $T' = (T + e) - e'$ with the same vertex set as T.

We now start the minimum spanning tree problem. Suppose $G = (V, E)$ is a connected graph in which every edge has a non-negative weight $w(e)$. The problem is to find a connected spanning subgraph having the minimum sum of edge weights. According to Lemma 11, this spanning subgraph is a spanning'tree for otherwise we may delete an edge from a cycle to decrease the sum of edge weighs.

The following is an algorithm for the minimum spanning tree problem.

Algorithm KRUSKAL.

1. sort the edge according to weights: $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_{|E|})$;
2. let $T = \emptyset$;
3. **for** $i = 1$ **to** $|E|$ **do**
4. **if** $T + e_i$ has no cycle **then** $T = T + e_i$;
5. **end.**

First, the correctness of the algorithm.

Theorem 16 *Algorithm KRUSKAL gives a minimum spanning tree of G.*

Proof. The final T is certainly a spanning subgraph of G without cycle. Suppose it is not connected. Then T has at least two components A and B. Choose x from A and y from B. Let $P : x, \ldots, x', y', \ldots, y$ be an x-y path in G, where y' is the first vertex not in A. It is the case that $x'y' \notin T$ and $T + x'y'$ also has no cycle. This is impossible as we then must add $x'y'$ at the iteration when we scan it at line 4 of the algorithm. This gives that T is a spanning tree of G.

Suppose G has a spanning tree T^* with sum of edge weights smaller than T. Let the edges of T are $e_{i_1}, e_{i_2}, \ldots, e_{i_{n-1}}$ chosen in the algorithm in this order. Let j be the first index such that $e_{i_j} \notin T^*$. We may assume that T^* is chosen so that j is as large as possible. Notice that $T^* + e_{i_j}$ has the fundamental cycle $C(T^*, e_{i_j})$, which must has an edge e not in T as T has no cycle. As $\{e_{i_1}, e_{i_2}, \ldots, e_{i_{j-1}}, e\} \subseteq T^*$ which has no cycles, it must be the case that e is scanned after e_{i_j} in the algorithm for otherwise e must be added into T. This gives that $w(e_{i_j}) \leq w(e)$ and then $T' = (T^* + e_{i_j}) - e$ gives a spanning tree with sum of edge weights no more than and hence equals to T^*. But the first index j' for which $e_{i_{j'}} \notin T'$ is larger than j, a contradiction to the choice of T^*. Hence T is in fact a minimun spanning tree of G. □

We now discuss the implementation of the algorithm.

The sorting for the edges costs $O(|E| \log |E|)$ time by using a heap sort.

There are several ways to implement the testing in line 4. We first describe them abstractly. We keep a partition of V, which correspond to the components of T. At beginning, the partition is $\{\{v\} : v \in V\}$ which corresponds to $T = \emptyset$. The UNION-FIND scheme is then used. namely, we needed two subroutines FIND(v) and UNION(s, t). FIND(v) returns the name of the set containing the vertex v, and UNION(s, t) replace the two sets s and t by their union. We then may implement line 4 as

assume $e_i = xy$; let $s =$ FIND(x) ; let $t =$ FIND(y) ;

if $s \neq t$ **then** { $T = T \cup \{e_i\}$; UNION(s,t) } ;

To implement the UNION-FIND system efficiently, we need to choose a proper data structure and use it in a good way. It is usually the case that when we want to save time for FIND then we need more time on UNION, and when we want to same time for UNION we very often need more time on FIND.

A simple minded implementation is to keep an array SET$[1..n]$, where SET$[i]$ is the name of the set containing vertex i. Initially, we set SET$[i] = i$. In this case, FIND$[x]$ and FIND$[y]$ take a constant time. For UNION$[s,t]$, we may scan the array SET$[1..n]$ and changing all SET$[i]$ with value from s to t. As we totally need to use $2|E|$ times of FIND and $|V|-1$ times of UNION, the time complexity is $O(|E|+|V|^2)$. This time is dominated by the sorting time $O(|E|\log|E|)$ if the graph is dense, i.e. $|E|$ is close to $|V|(|V)-1)/2$. However, when the graph is sparse, i.e. $|E|$ is close to $|V|$, the sorting time is only $O|V|\log|V|)$, which is much smaller than $(|V|^2)$.

To improve the implementation, beside the array SET$[1..n]$ we may keep each set in a linked list and keep the size of the list called SIZE$[s]$. To perform a UNION(s,t), we may assume SIZE$[s]$ $\leq$SIZE$[t]$ otherwise interchange the role of s and t. We then merge the two lists in a constant time, and scan the list for set s and set SET$[i]$ to be t for all i in set s. It can be proved that the number of times to rename the name of the set containing a vertex is at most $\log|V|$. Thus, the time complexity is now $O(|E|+|V|\log|V|)$.

Homework 8 *Prove that in the implementation above, the number of times to rename the name of the set containing a vertex is at most* $\log|V|$.

Although, any improvement on the UNION-FIND system is of no use as the sorting part already needs $O(|E|\log|E|)$ time, for other purpose, people do study different data structure and implementation. See the book by Aho et al.

The following algorithm is often used for the case when the graph is dense.

Algorithm PRIM

```
1.    let d(y) = w(x,y) and p(y) = x for all vertices y ≠ x ;
2.    let Q = {x} ; let T = ∅ ;
3.    while (Q ≠ V)
4.    {   choose a vertex v not in Q with minimum d(v) ;
5.        Q = Q ∪ {v} ; T = T ∪ {vp(v)} ;
6.        for all vertices u not in Q do
7.            if w(v,u) < d(u) then {d(u) = w(v,u); p(u) = v; } ;
8.    }
```

The algorithm is nearly identical to Dijkstra's algorithm for the shortest path problem. It takes $O(|V|^2)$ time.

Homework 9 *If C is a cycle in a graph $G = (V,E)$ and $A \subseteq V$, then C has an even number of edges with one end-vertex in A and the other not in A.*

Homework 10 *Prove that Algorithm PRIM correctly gives a minimum spanning tree of a connected graph.*

A more general theorem can be used to cover the above two algorithms is as follows.

Homework 11 *Suppose $G' = (V, E')$ is an acyclic graph which can be extended to a minimum spanning tree of a connected graph $G = (V, E)$. If A is a component of G' and among all edges with one end-vertex in A and the other not in A the edge xy is one with the minimum weight, then $G' + xy$ can be extended to a minimum spanning tree of G.*

March 17, 2005

7 Cayley's theorem

There are many ways to prove the following theorem by Cayley. Here we give an algorithmic proof, which is due to Prüfer.

Theorem 17 (Cayley) *The number of spanning trees for the complete graph with n distinct vertices is n^{n-2}.*

Proof. Let $V = \{1, 2, \ldots, n\}$. We shall give a one-to-one correspondence between the set of all spanning trees and the n^{n-2} words of length $n-2$ over the alphabet V. The algorithm for finding the word which corresponds to a given tree is as follows:

1. **for** $i = 1$ **to** $n - 2$ **do**;
2. among all leaves of the current tree let j be the least one ;
3. delete j and its incident edge e from the tree;
4. the ith letter of the word is the other end-vertex of e;
5. **end.**

Notice that the mapping is well-defined by Lemma 13. To see the mapping is a bijection, we consider another mapping which is the inverse of this one.

Let $w = a_1 a_2 \ldots a_{n-2}$ be a word over V. If T is the tree for which the algorithm produces w then the degree $d(k)$ of a vertex k in T is equal to 1 plus the number of times k appears in w. This follows from the observation that when each, but the last, of the edges incident to k is deleted, k is written as a letter of w. For instance, if $w = 4164$ then $d(1) = 2, d(2) = 1, d(3) = 1, d(4) = 3, d(5) = 1$ and $d(6) = 2$ in the tree which produced w.

The reverse of the mapping is now as follows:

1. **for** $i = 1$ **to** $n - 2$ **do**;
2. let j be the least vertex for which $d(j) = 1$;
3. construct an edge ja_i ; $d(j) = 0$; $d(a_i) = a(a_i) - 1$;
4. **end.**
5. construct an edge between the two vertices whose degree is 1.

It can be proved that the above two mapping are inverse to each other, and hence are two bijections. □

We close this section by remark that there is in fact a formula for the number of spanning trees for a general graph. Suppose $G = (V, E)$ an multi-graph of n vertices. Consider its *degree matrix* which is the $n \times n$ matrix D with

$$D(i,j) \;=\; \begin{cases} \deg(i) & \text{if } i = j; \\ -k & i \neq j \text{ and } k \text{ is the number of edges between } i \text{ and } j. \end{cases}$$

Theorem 18 *The number of spanning trees of a multi-graph is equal to the minors of its degree matrix which results from the erasure of a row and a corresponding column.*

We may apply this theorem to calculate the number of spanning trees of the complete graph K_n. In this case, the degree matrix is the following $n \times n$ matrix.

$$\begin{pmatrix} n-1 & -1 & \dots & -1 \\ -1 & n-1 & \dots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & \dots & n-1 \end{pmatrix}.$$

After erasing one row and the corresponding column, the matrix looks the same except that it is now $(n-1) \times (n-1)$. We can now add to any column (or row) a linear combination of the others, without changing its determinant. First subtract the first column from every other. We get:

$$\begin{pmatrix} n-1 & -n & -n & \dots & -n \\ -1 & n & 0 & \dots & 0 \\ -1 & 0 & n & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -1 & 0 & 0 & \dots & n \end{pmatrix}.$$

Now add every one of the other rows to the first:

$$\begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ -1 & n & 0 & \dots & 0 \\ -1 & 0 & n & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -1 & 0 & 0 & \dots & n \end{pmatrix}.$$

Clearly, the determinate of this matrix is n^{n-2}.

March 22, 2005

8 Huffman trees

Suppose we have n sources letters having probability $p_1, p_2, \ldots, p_n$ of appearance. The goal is to construct a prefix code for these letters, which correspond to a binary tree of n leaves. We also want the average code-word length

$$\overline{\ell} = \sum_{i=1}^{n} p_i \ell_i$$

to be as small as possible, where ℓ_i is the length of the ith code-word which is the length from the the corresponding leaf of the binary tree to its root.

Let $P = \{p_1 \geq p_2 \geq \ldots \geq p_n\}$ and the corresponding optimal binary tree is $T(P)$. Then it is the case that the leaves corresponding to p_{n-1} and p_n are of two fairest leaves to the root among all leaves, otherwise we may interchanging them to those with smaller p_i to reduce $\overline{\ell}$. And then in fact we may assume that they are the children of the same parent x. Deleting these two leaves we get a binary tree in which x is a leaf. Use this as a binary tree for $P' = \{p'_1, p'_2, \ldots, p'_{n-1}\}$ where $p'_i = p_i$ for $1 \leq i \leq n-2$ and $p'_{n-1} = p_{n-1} + p_n$. Then

$$\overline{\ell} = \sum_{i=1}^{n} p_i \ell_i = \sum_{i=1}^{n-2} p'_i \ell'_i + p' \geq \overline{\ell'} + p'_{n-1}.$$

On the other hand, suppose we have an optimal binary tree for P'. Add two leaves as the children of the leaf corresponding to p'_{n-1}, we get a binary tree for P. Then

$$\overline{\ell'} + p'_{n-1} = \sum_{i=1}^{n-2} p'_i \ell'_i + p' = \sum_{i=1}^{n} p_i \ell_i \geq \overline{\ell}.$$

Hence $\overline{\ell} =\geq \overline{\ell'} + p'_{n-1}$. We then have an algorithm by merging the two smallest probabilities recursively to produce an optimal tree.

9 Matroids

As we saw that the greedy algorithm works for the minimum spanning tree problem. However, it is not always the case that the greedy algorithm works for any kind of optimization problem. For instance, consider the maximum weighted matching problem as follows. A *matching* in a graph is a subset M of edges in which no two edges have a common endvertex. For a graph in which every edge e has a positive weight $w(e)$, the problem is to find a matching M such that $\sum_{e \in M} w(e)$ is maximum. Consider the 4-cycle $C_4 = (V, E)$ where $V = \{1, 2, 3, 4\}$ and $E = \{12, 23, 34, 41\}$. Let $w(12) = 10, w(23) = w(41) = 7$ and $w(34) = 1$. While the greedy algorithm produces $\{12, 34\}$, the only maximum matching is $\{23, 41\}$.

The secrete behinds the greedy algorithm also appears in many other places, which was made as an abstract concept called matroid by Whitney in 1930s.

As will been seen, a matroid may be defined in many different but equivalent ways, several of which were described in Whitney's original paper. Deciding which set of axioms would be the most natural to start with was difficult. I decide to settle the "dependency relation" because I think it is most nature to verify graphical matroids and vectorial matroids.

A *matroid* is an ordered pair $M = (S, \sim)$, where S is a finite set S and $\sim$ is a relation between S and 2^S such that for $A, B \subseteq S$ and $x, y \in S$ the following conditions hold.

(D1) If $x \in A$, then $x \sim A$.
(D2) If $x \sim A$ and $y \sim B$ for all $y \in A$, then $x \sim B$.
(D3) If $y \not\sim A$ but $y \sim A \cup x$, then $x \sim A \cup y$.

Example 1. (Graphical matroid.) Suppose $G = (V, E)$ is a pseudo-graph. Let $S = E$; and for any $xy \in$ and $A \subseteq S$ define $xy \sim A$ as there is an x-y walk in (V, A).

Example 2. (Vectorial matroid.) Let S be a finite subset of a vector space; and for any $x \in S$ and $A \subseteq S$ define $x \sim A$ as that x is a linear combination of elements in A.

Example 3. (Uniform matroid.) Suppose $n \geq k$ are positive integers. The uniform matroid $U_{k,n}$ is one with $S = \{1, 2, \ldots, n\}$; and $x \sim A$ is and only if $x \in A$ or $|A| \geq k$.

Notice that a graphical matroid is in fact a vectorial matroid. Suppose the matroid comes from a pseudo-graph $G = (V, E)$. Consider its edge-vertices incidence matrix as over Z_2. Then the correspondence matroid is in fact the same as the vectorial matroid whose ground set is the set of all column vectors of the edge-vertex incidence matrix.

Homework 12 *Find a matroid which is not vectorial.*

Homework 13 *Prove that the uniform matroid $U_{2,4}$ is not isomorphic to any matroid which is vectorial over Z_2, but is isomorphic to some vectorial matroid over Z_3.*

Following the analogy with graphs and vector spaces we make the following definitions. Suppose $M = (S, \sim)$ is a matroid.

The *closure* σA of a subset A of S is the set of all $x \sim A$.

A subset D of S is *dependent* if $x \sim D - x$ for some $x \in D$. A *circuit* is a minimal dependent set. The collection of all circuits in M is denoted by $\mathcal{C}$ or $\mathcal{C}(M)$.

A subset I of S is *independent* if $x \not\sim I - x$ for all $x \in I$. The collection of all independent sets in M is denoted by $\mathcal{I}$ or $\mathcal{I}(M)$.

A *base* of M is a maximal independent subset of S. The collection of all bases in M is denoted by $\mathcal{B}$ or $\mathcal{B}(M)$.

The *rank* $\rho(A)$ of a subset A of S is the maximum size of an independent subset of A.

Homework 14 *Determine the above terms for the matroids in Examples* 1 *to* 3.

March 22, 2005

Theorem 19 (Closure axioms) *A function $\sigma : 2^S \to 2^S$ is the closure operator of a matroid on S if and only if for any $A, B \subseteq S$ and $x, y \in S$ the following conditions hold.*

(S1) $A \subseteq \sigma A$.
(S2) *If* $A \subseteq B$, *then* $\sigma A \subseteq \sigma B$.
(S3) $\sigma A = \sigma\sigma A$.
(S4) *If* $y \notin A, y \in \sigma(A \cup x)$, *then* $x \in \sigma(A \cup y)$.

Theorem 20 (Independency axioms) *A collection $\mathcal{I}$ of subsets of S is the set of independent sets of a matroid on S if and only if the following conditions hold.*

(I1) $\emptyset \in \mathcal{I}$.
(I2) *If* $A \subseteq B \in \mathcal{I}$, *then* $A \in \mathcal{I}$.
(I3) *If* $A, B \in \mathcal{I}$ *and* $|A| < |B|$, *then* $A \cup y \in \mathcal{I}$ *for some* $y \in B - A$.

Theorem 21 (Base axioms) *A non-empty collection $\mathcal{B}$ of subsets of S is the set of bases of a matroid on S if and only if the following conditions hold.*

(B1) *If* $B_1, B_2 \in \mathcal{B}$ *and* $x \in B_1 - B_2$, *then* $B_1 - x \cup y \in \mathcal{B}$ *for some* $y \in B_2 - B_1$.

Theorem 22 (Base axioms) *A non-empty collection $\mathcal{B}$ of subsets of S is the set of bases of a matroid on S if and only if the following conditions hold.*

(B2) *If* $B_1, B_2 \in \mathcal{B}$ *and* $x \in B_1 - B_2$, *then* $B_2 \cup x - y \in \mathcal{B}$ *for some* $y \in B_2 - B_1$.

Theorem 23 (Rank axioms) *A function $\rho : 2^S \to Z$ is the rank function of a matrod on S if and only if for any $A \subseteq S$ and $x, y \in S$ the following conditions hold.*

(R1) $\rho\emptyset = 0$.
(R2) $\rho A \leq \rho(A \cup x) \leq \rho A + 1$.
(R3) $\rho(A \cup x) = \rho(A \cup y) = \rho A$, *then* $\rho(A \cup x \cup y) = \rho A$.

Theorem 24 (Rank axioms) *A function $\rho : 2^S \to Z$ is the rank function of a matrod on S if and only if for any $A, B \subseteq S$ the following conditions hold.*

(R1') $0 \leq \rho A \leq |A|$.
(R2') *If* $A \subseteq B$, *then* $\rho A \subseteq \rho B$.
(R3') $\rho(A \cup B) + \rho(A \cap B) \leq \rho A + \rho B$.

Theorem 25 (Circuit axioms) *A collection $\mathcal{C}$ of subsets of S is the set of circuits of a matroid on S if and only if the following conditions hold.*

(C1) *If* $C_1, C_2 \in \mathcal{C}$ *with* $C_1 \neq C_2$, *then* $C_1 \not\subseteq C_2$.
(C2) *If* $C_1, C_2 \in \mathcal{C}$ *with* $C_1 \neq C_2$ *and* $x \in C_1 \cap C_2$, *then* $C_3 \subseteq (C_1 \cup C_2) - x$ *for some* $C_3 \in \mathcal{C}$.

10 Properties for independent sets

This section is devoted for properties on independent sets. In particular, we shall prove Theorem 20. To make the proof easy, we first give two useful properties which will be used frequently.

(Da) If $x \sim A \subseteq B$, then $x \sim B$.

Proof. For any $y \in A \subseteq B$, by (D1), $y \sim B$. The property then follows from (D2). □

(Db) If $A \in \mathcal{I}$ but $A \cup x \notin \mathcal{I}$, then $x \sim A$.

Proof. Since $A \cup x \notin \mathcal{I}$, there is some $y \in A \cup x$ such that $y \sim (A \cup x) - y$. If $y = x$, then $x \sim A$. Otherwise, $y \in A$ and then $A \in \mathcal{I}$ implies $y \not\sim A - y$. As $y \sim (A - y) \cup x$, by (D3), we again have $x \sim (A - y) \cup y = A$. □

Now we are ready to prove Theorem 20. Given a matrod $M = (S, \sim)$, we first prove that (I1) to (I3) hold.

(I1) obviously holds.

To see (I2), suppose $A \subseteq B \in \mathcal{I}$ but $A \notin \mathcal{I}$. Then, $x \sim A - x \subseteq B - x$ for some $x \in A$. By (Da), $x \sim B - x$, contradicting that $B \in \mathcal{I}$. Hence, $A \in \mathcal{I}$.

To see (I3), suppose it is not true. Let A and B be chosen so that $|A| < |B|$ and $|A \cup B|$ is maximum. Clearly, $B \not\subseteq A$, so take $y \in B - A$. Suppose there is some $x \in A - B$ such that $B' = (B - y) \cup x \in \mathcal{I}$. In this case, $|A| < |B| = |B'|$ and $|A \cap B| < |A \cap B'|$. By the choice of A and B, there is some $y' \in B' - A \subseteq B - A$ such that $A \cup y' \in \mathcal{I}$, and we are done. Hence, $(B - y) \cup x \notin \mathcal{I}$ for all $x \in A - B$. Since $B - y \in \mathcal{I}$, by (Db), $x \sim B - y$ for all $x \in A - B$ and hence for all $x \in A$. Then, $y \not\sim A$ for if $y \sim A$ then by (D2) $y \sim B - y$, contradicting that $B \in \mathcal{I}$. This further implies that $A \cup y \in \mathcal{I}$ for otherwise $A \cup y \notin \mathcal{I}$ and $A \in \mathcal{I}$ lead to $y \sim A$ by (Db), which is impossible.

March 29, 2005

Conversely, suppose we start from a finite set S with a family $\mathcal{I} \subseteq 2^S$ such that (I1) to (I3) hold. Now define a relation between S and 2^S as:

$$x \sim A \text{ if and only if } x \in A \text{ or } I \cup x \notin \mathcal{I} \text{ for some } I \subseteq A \text{ with } I \in \mathcal{I}.$$

We shall prove that $M = (S, \sim)$ is a matroid with $\mathcal{I}(M) = \mathcal{I}$.

(D1) follows from the definition of $\sim$.

To see (D2), suppose $x \sim A$ and $y \sim B$ for all $y \in A$. For the case when $x \in A$, we have $x \sim B$. For the case when $x \notin A$, by the definition of $\sim$, we have $I \cup x \notin \mathcal{I}$ for some $I \subseteq A$ with $I \in \mathcal{I}$. If $I \subseteq B$, then $x \sim B$. Otherwise there is some $y \in I - B \subseteq A - B$. So, $y \sim B$ and $y \notin B$ imply that ... (to be filled).

To see (D3), suppose $y \not\sim A$ but $y \sim A \cup x$. For the case of $y = x$, we have $x \sim A \cup y$. Now suppose that $y \neq x$. The condition $y \sim A \cup x$ implies that either $y \in A \cup x$ or $I \cup y \notin \mathcal{I}$

for some $I \subseteq A \cup x$ with $I \in \mathcal{I}$. The former implies that $y \in A$ and so $y \sim A$, which is impossible. The latter implies that $x \in I$ for otherwise $I \subseteq A$ gives $y \sim A$, again is impossible. Then, $(I - x \cup y) \cup x \notin \mathcal{I}$ with $I - x \cup y \subseteq A \cup y$ and $I - x \cup y \in \mathcal{I}$. This gives that $x \sim A \cup y$ as desired.

Finally, we need to show that $\mathcal{I} = \mathcal{I}(M)$.

Suppose $A \in \mathcal{I}$. For any $x \in A$, we have $x \notin A - x$ and $I \cup x \in \mathcal{I}$ for all $I \subseteq A - x$. Notice that the later follows from (I2) as $I \cup x \subseteq A \in \mathcal{I}$. This gives that $x \not\sim A - x$ for all $x \in A$, or $A \in \mathcal{I}(M)$.

Suppose $A \notin \mathcal{I}$. Choose a maximal subset I of A with $I \in \mathcal{I}$. There must exist some $x \in A - I$. Then, $I \cup x \notin \mathcal{I}$ and so $x \sim A - x$ or $A \notin \mathcal{I}(M)$.

Homework 15 *The conditions* (I3) *in the independency axioms can be replaced by:*
(I3') *If I and J are two maximal independent subsets of a set $A \subseteq S$, then $|I| = |J|$.*

We close this section by proving the correctness of condition (B1) and (B2) in the base axioms.

We first prove (B1). Suppose $B_1, B_2 \in \mathcal{B}$ and $x \in B_1 - B_2$. By the homework above we know that $|B_1 - x| = |B_1| - 1 = |B_2| - 1$. According to (I3), $B_1 - x \cup y \in \mathcal{I}$ for some $y \in B_2 - (B_1 - x) = B_2 - B_1$. Extend $B_1 - x \cup y$ to a base B. Again by the homework above we have

$$|B_1| = |B_1 - x \cup y| \leq |B| = |B_1|,$$

and so in fact $B_1 - x \cup y = B \in \mathcal{B}$.

We next prove (B2). Suppose $B_1, B_2 \in \mathcal{B}$ and $x \in B_1 - B_2$. As $(B_1 \cap B_2) \cup x \subseteq B_1$, we have that $(B_1 \cap B_2) \cup x$ independent. We may extend it to a maximal independent set B of $B_2 \cup x$. First $|B| \leq |B_2|$. Also, as B_2 is an independent subset of $B_2 \cup x$, we have $|B_2| \leq |B|$ and so $|B| = |B_2|$. It is then the case that $B = B_2 \cup x - y$ for some $y \in B_2 - B_1$.

11 greedy algorithm in matroids

Suppose M is a matroid in which every element e is associated with a positive real number $w(e)$. The problem is to find an independent set I with $w(I) = \sum_{e \in I} w(e)$ maximum. Notice that an optimal solution I must be a base.

Algorithm GREEDY.
1. sort the elements according to weights: $w(e_1) \geq w(e_2) \geq \ldots \geq w(e_m)$;
2. let $I = \emptyset$;
3. **for** $i = 1$ **to** m **do**
4. **if** $I + e_i \in \mathcal{I}$ **then** $I = I + e_i$.

To prove that the greedy algorithm works, we first observe that the final output $I = \{e_{j_1}, e_{j_2}, \ldots, e_{j_r}\}$ with $j_1 < j_2 < \ldots < j_r$ is a base. Suppose to the contrary I is not a base.

There must exist some $e_i \notin I$ such that $I \cup e_i$ is independent, where $j_k < j < j_{k+1}$. Then, $\{e_{j_1}, e_{j_2}, \ldots, e_{j_k}, e_i\}$ is independent. When the algorithm runs at iteration i, it must be the case that e_i is put into I, a contradiction to that $e_i \notin I$.

Suppose I is not a maximum weighted base. Choose a base B with $w(B) > w(I)$. Let s be the first index such that $e_{j_s} \in I - B$ but $\{e_{j_1}, e_{j_2}, \ldots, e_{j_{s-1}}\} \subseteq I \cap B$. Without loss of generality, we may assume that B is chosen so that s is as large as possible. By the axioms for bases, there is some $e_i \in B - I$ such that $B' = B \cup e_{j_s} - e_i \in \mathcal{B}$. As $\{e_{j_1}, e_{j_2}, \ldots, e_{j_s}\} \subseteq I \cap B'$, it is the case that the s' for B' is larger than s. By the choice of B, it is then the case that $w(I) \geq w(B')$. Then $w(B) > w(B') = w(B) + w(e_{j_s}) - w(e_i)$ implies that $w(e_i) > w(e_{j_s})$ and so $i < j_s$. However, $\{e_{j_1}, e_{j_2}, \ldots, e_{j_{s-1}}, e_i\}$ is a subset of B and hence is independent. When the algorithm run at iteration i, the element e_i must be chosen, a contradiction. □

March 31, 2005

Theorem 26 *Suppose S is a finite non-empty set and $\mathcal{I} \subseteq 2^S$ such that* (I1) *and* (I2) *hold. If the greedy algorithm gives a maximum weighted base for all positive weight function w, then* (I3) *holds.*

Proof. Suppose $A, B \in \mathcal{I}$ and $|A| < |B|$. First, $|B - A| > |A - B| \geq 0$. Choose a positive real number ϵ such that $2\epsilon|S| < 1$. Consider the weight function w defined by

$$w(x) = \begin{cases} 1+\epsilon & \text{if } x \in A; \\ 1 & \text{if } x \in B - A; \\ \epsilon & \text{if } x \in S - (A \cup B). \end{cases}$$

The greedy algorithm gives an optimal base I with $A \subseteq I$. Now, $w(I) - w(A) \geq w(B) - w(A) = |B-A| - (1+\epsilon)|A-B| \geq 1 - \epsilon|S|$. If $I \subseteq S - B$, then $w(I) - w(A) \leq \epsilon|S - (A \cup B)| < \epsilon|S| < 1 - \epsilon|S|$, which is impossible. So there is some $y \in (B - A) \cap I$, which gives that $A \cup y \in \mathcal{I}$. □

12 Bipartite matching

In a graph $G = (V, E)$, a *matching* is a subset $M \subseteq E$ such that every two distinct edges in M have no common end vertex.

As a maximal matching is not necessary a maximum sized matching, for instance $\{23\}$ in P_4, the set of matching does not form the family of independent sets of a matroid.

The famous problem on SDR is equivalent to the problem of finding a maximum matching in a bipartite graph. Here we assume that $G = (V, E)$ is a bipartite graph whose vertex set V can be partition into X and Y such that every edge has one end vertex in X and the other in Y. Let $\mathcal{I}$ denote the set of all matchings in G. Although $(E, \mathcal{I})$ is not always a matroid, it is the intersection of two matroids $(E, \sim_X)$ and $(E, \sim_Y)$.

Homework 16 *Suppose $G = (V, E)$ is a bipartite graph in which $X \cup Y$ is a bipartition of V. For $xy \in E$, where $x \in X$ and $y \in Y$, and $A \subseteq E$, define $xy \sim_X A$ as $xy \in A$ or $xy' \in A$ for some $xy' \in A$. Prove that $(E, \sim_X)$ is a matroid.*

Notice that in the homework above, a set A is independent if and only if every two distinct edges in A have no common end vertex in X. If we define $\sim_Y$ similarly, then A is a matching if and only if it is $\sim_X$-independent and $\sim_Y$-independent.

The algorithm in this section for finding a maximum matching need the following terms.

For a matching M, an *M-alternating path* is a path

$$v_0, e_1, v_1, e_2, v_2, e_2, \ldots, e_r, v_r,$$

where the edges alternatively appear and does not appear in M. A vertex is *M-saturated* if it is the end vertex of some edge in M; and *M-exposed* otherwise. An *M-augmenting path* is an M-alternative path whose two end vertices are M-exposed. Note that for an M-augmenting path, it is the case that r is even and $e_i \notin M$ for odd i and $e_i \in M$ for even i.

Lemma 27 *If M is a matching and P is an M-augmenting path, then $(M - E(P)) \cup (E(P) - M)$ is a matching of size one greater than M.*

The main idea in the algorithm for the maximum bipartite matching problem in this section is to start with the empty matching and finding an augmenting path to update the matching. Although the following lemma is true, we does not use it to prove the correctness of the algorithm.

Lemma 28 (Berge) *In a graph G, a matching M is maximum if G has no M-augmenting path.*

Homework 17 *Prove the lemma above. Notice that we do not assume that G is bipartite in the lemma.*

Algorithm BIPARTITE-MATCHING.

1. $M = \emptyset$;
2. erase all labels;
3. label all M-exposed vertices u in X by "0" and put them into Q ;
4. **while** $(Q \neq \emptyset)$
5. { enqueue a vertex u from Q;
6. **if** $u \in X$ **then** label all un-labeled neighbors v of u by "u" and put them into Q ;
7. **if** $u \in Y$ **then if** u is M-exposed
8. **then** an M-augmenting path is found, update M and **goto** line 2 ;
9. **else** detect $uv \in M$, label v by "u" and put it into Q;
10. }

April 7, 2005

The algorithm is attempt to find all possible M-augmenting path as in the while loop. However, it is not the case that we may conclude that there is no M-augmenting path when the algorithm stops. This is why we can not use Berge's lemma to verify the correctness of the algorithm. Instead we shall use the following primal-dual approach to prove the correctness of the algorithm.

A *vertex cover* of a graph $G = (V, E)$ is a vertex set $C \subseteq V$ such that for any edge $xy \in E$ either $x \in C$ or $y \in C$. For any matching M and any vertex cover C, it is the case that we can associate with each edge in M an end vertex in C, while the mapping is one-to-one as no two distinct edges in M have a common end vertex. This gives the following *weak duality inequality*:

$$\max\{|M| : M \text{ is a matching in } G\} \leq \min\{|C| : C \text{ is a vertex cover of } G\}.$$

We in fact shall prove that the equality holds for any bipartite graph.

Suppose the algorithm stops with the final matching M^*. Let L consist of all labeled vertices; and $C^* = (X - L) \cup (Y \cap L)$. We claim that C^* is a vertex cover. Suppose to the contrary that it is not. Then there is some edge xy with $x \in X$ and $y \in Y$ such that $x \in L$ and $y \notin L$. But this is impossible as line line 6, if x is labeled, then all its un-labeled neighbors including y must be labeled. Thus, C^* is a vertex cover.

We next show that $|C^*| \leq |M^*|$. To see this, for any $u \in C^*$ we shall associate to it with an edge in M^* as follows. For the case when $u \in X$, it is un-labeled and so it is M^*-saturated by some $uv \in M^*$. Note that in this case v is also un-labeled for otherwise lines 7 and 9 force v to be labeled. For the case of $u \in Y$, it is labeled and so by lines 7 and 9 again it corresponds to an edge $uv \in M^*$ and v is labeled. As a un-labeled (respectively, labeled) vertex is associated with an edge in M^* whose two end vertices are all un-labeled (respectively, labeled), the this mapping is one-to-one, which gives $|C^*| \leq |M^*|$. Hence,

$$\begin{aligned} |M^*| &\leq \max\{|M| : M \text{ is a matching in } G\} \\ &\leq \min\{|C| : C \text{ is a vertex cover of } G\} \\ &\leq |C^*| \\ &\leq |M^*|. \end{aligned}$$

Consequently, all inequality are equalities. In other words, we have the *strong duality equality*:

$$\max\{|M| : M \text{ is a matching in } G\} = \min\{|C| : C \text{ is a vertex cover of } G\}.$$

Also, we know that M^* is a maximum matching as desired, and C^* is a minimum vertex cover as a by-product.

13 Weighted bipartite matching

We may associate with every edge e of a bipartite graph a real number $w(e)$ and ask the problem for finding a matching M with a maximum weight $w(M) = \sum_{e \in M} w(e)$. The

problem can also be solved by a similar method except now weights are counted. The approach is still a primal-dual method, which has a flavor as in the linear programming.

We shall use the notion that an edge ij with $i \in X$ and $j \in Y$ has the weight w_{ij}. A *w-vertex cover* is a vector(u, v) where each vertex $i \in X$ has a non-negative real number u_i and each vertex $j \in Y$ a non-negative real number v_j such that $w_{ij} \leq u_i + v_j$ for each edge $ij \in E$. Let the *cost* of (u, v) be $\text{cost}(u, v) = \sum_{i \in X} u_i + \sum_{j \in Y} v_j$. We then have

$$w(M) = \sum_{ij \in M} w_{ij} \leq \sum_{ij \in M} (u_i + v_j) \leq \sum_{i \in X} u_i + \sum_{j \in Y} v_j = \text{cost}(u, v).$$

Also, the inequality is an equality if any only if the following conditions hold.

(CS1) If $ij \in M$, then $w_{ij} = u_i + v_j$.

(CS2) If $i \in X$ is M-exposed, then $u_i = 0$.

(CS3) If $j \in Y$ is M-exposed, then $v_j = 0$.

Consequently, we have the *weak duality inequality*:

$$\max\{w(M) : M \text{ is a matching}\} \leq \inf\{\text{cost}(u, v) : (u, v) \text{ is a } w\text{-vertex cover}\}.$$

Notice that we use inf instead of min because there are infinitely many w-vertex covers.

The approach we use in this section is to keep a matching M and a w-vertex cover (u, v) so that (CS1) and (CS3) always hold. The algorithm keeps the number of $i \in X$ such that (CS2) fails decreases. At the end of the algorithm, we get a matching M^* and a w-vertex cover (u^*, v^*) with (CS1) to (CS3) hold. And so in fact $w(M^*) = \text{cost}(u^*, v^*)$. We then have

$$\begin{aligned} w(M^*) &\leq \max\{w(M) : M \text{ is a matching}\} \\ &\leq \inf\{\text{cost}(u, v) : (u, v) \text{ is a } w\text{-vertex cover}\} \\ &\leq \text{cost}(u^*, v^*) \\ &= w(M^*) \end{aligned}$$

Therefore, all inequalities are equalities. Notice that by now we may replace inf by min because the inf attends by $\text{cost}(u^*, v^*)$. So, we have three conclusions. First, M^* is a maximum weighted matching as we want. Second, (u^*, v^*) is a minimum w-vertex cover. Finally, we have the *strong duality equality*.

$$\max\{w(M) : M \text{ is a matching}\} = \min\{\text{cost}(u, v) : (u, v) \text{ is a } w\text{-vertex cover}\}.$$

April 12, 2005

To write the algorithm, we give each vertex $j \in Y$ a value π_j to store $u_i + v_j - w_{ij}$ for an edge $ij \in M$. So, condition (CS1) is the same as

(CS1') If $ij \in M$, then $\pi_j = 0$.

Now we are ready to give the algorithm. Notice that in the following algorithm (CS1) and (CS3) always hold, and the number of $i \in X$ for which (CS2) fails decreases by one whenever line 8 or line 9 is excused.

Algorithm WEIGHTED-BIPARTITE-MATCHING.

1. $M = \emptyset$;
   $u_i = \max\{0, \max_{ij \in E} w_{ij}\}$ for all $i \in X$;
   $v_j = 0$ and $\pi_j = \infty$ for all $j \in Y$;
2. **if** there is no M-exposed vertex i with $u_i > 0$ **then STOP** ;
3. choose an M-exposed verex i with $u_i > 0$ and label it "$\emptyset$" ;
4. choose a un-scanned vertex i with label, where
   either $i \in X$ (**goto** line 5 in this case)
   or $i \in Y$ with $\pi_i = 0$ (**goto** line 6 in this case) ;
   goto line 7 if there is no such vertex ;
5. scan the labeled vertex $i \in X$ as follows:
   for all $ij \in E - M$ with $u_i + v_j - w_{ij} < \pi_j$ **do**
   label j by "i" and set $\pi_j = u_i + v_j - w_{ij}$;
   goto line 4 ;
6. scan the labeled vertex $i \in Y$ with $\pi_i = 0$ as follows:
   if i is exposed **then goto** line 9 ;
   else identify $j \in X$ such that $ij \in M$ and label j by "i" ;
   goto line 4 ;
7. $\delta_1 = \min\{u_i : i \in X \text{ labeled}\}$;
   $\delta_2 = \min\{\pi_j : j \in Y \text{ with } \pi_j > 0\}$;
   $\delta = \min\{\delta_1, \delta_2\}$;
   $u_i = u_i - \delta$ for all labeled $i \in X$;
   $v_j = v_j + \delta$ for all labeled $j \in Y$ with $\pi_j = 0$;
   $\pi_j = \pi_j - \delta$ for all labeled $j \in Y$ with $\pi_j > 0$;
   if $\delta_1 > \delta_2$ **then goto** line 4 **else goto** line 8 ;
8. choose a labeled vertex $i \in X$ with $u_i = 0$;
   back trace from i through labels to get an M-alternating path P ;
   replace M by $(M - P) \cup (P - M)$;
   remove all labels from vertices and set $\pi_j = \infty$ for all $j \in Y$;
   goto line 2;
9. back trace from $i \in Y$ with $\pi_i = 0$ to get an M-augmenting path P ;
   replace M by $(M - P) \cup (P - M)$;
   remove all labels from vertices and set $\pi_j = \infty$ for all $j \in Y$;
   goto line 2 ;

14 Depth-first search

Depth-first search is a powerful tool in the design and analysis of algorithms. It is useful to the problems of finding 2-connected components of a graph, finding strongly connected components of a digraphs, testing if a graph is planar ... etc. We start with the problem of finding all 2-connected components of a graph.

Consider visiting the vertices of a graph in the following manners. We select and

"visit" a starting vertex v. Then we select any edge vw incident upon v and visit w. In general, suppose x is the most recently visited vertex. The search continued by selecting some unexplored edge xy incident upon x. If y has been previously visited, we find another new edge incident upon x. If y has not been previously visited, then we visit y and begin the search anew starting at vertex y. After completing the search through all paths beginning at y, the search returns to x, the vertex from which y was first reached. The process of selecting unexplored edges incident upon x is continued until the list of these edges is exhausted. The method of visiting the vertices of a graph is called a *depth-first search* since we continue searching in the forward (deeper) direction as long as possible.

Depth-first search can be applied to a digraph as well. In this case, at vertex x we search only edge xy directed out of x. After exhausting all edges out of y, we return to x even through there may be other edges directed into y which have not yet been searched.

If depth-first search is applied to a graph which is connected, then it is easy to show that every vertex will be visited and every edge examined. If the graph is not connected, then a component of the graph will be searched. Upon completion of a component, a vertex not yet visited is selected as the new start vertex and a new search is begun.

A depth-first search of a graph $G = (V, E)$ partitions the edge into two sets T and B. An edge vw is placed in the set T if vertex w has not been previously visited when we are at vertex v considering edge vw. Otherwise, edge vw is placed in the set B. The edges in T are called *tree edges*, and those in B *back edge*. The subgraph (V, T) is a forest, called a *depth-first spanning forest for* G. In the case when G is connected, the forest consists of a single tree and (V, T) is called a *depth-first spanning tree*. We consider each tree in a depth-first search spanning forest to be rooted at that vertex at which the depth-first search of that tree was begun.

Algorithm DEPTH-FIRST-SEARCH.

```
1.  T = ∅ ;
2.  for all v in V do mark v "new" ;
3.  while there exists a vertex v in V marked "new" do
4.        SEARCH(v).

    procedure SEARCH(v) ;
5.     mark v "old" ;
6.     for each vertex w ∈ N(v) do
7.        if w is marked "new" then
8.        {  add vw to T ;
9.           SEARCH(w)
10.       }
```

All edges in E not placed in T are considered to be in B. Note that if edge vw is in E, then w will be in $N(v)$ and v will be in $N(w)$. Thus we cannot simply place edge vw in B if we are at vertex v and w is marked "old" since w might be the parent of v. To find the back edges properly, if necessary we need a parameter PARENT[v] for each vertex v.

Lemma 29 *If vw is a back edge, then in the spanning forest v is an ancestor of w or vice versa.*

Proof. Suppose without loss og generality that v is visited before w, in the sense that SEARCH(v) is called before SEARCH(w). Thus when v is reached, w is still labeled "new." All "new" vertices visited by SEARCH(v) will become descendants of v in the spanning forest. But SEARCH(v) cannot end until w reached, since w is in $N(v)$. □

There is a natural order which depth-first search imposes on the vertices of a spanning forest. Namely, vertices can be labeled in the order they are visited if we initialize COUNT to 1 between lines 1 and 2 of Algorithm DEPTH-FIRST-SEARCH and insert

DEFNUMBER[v] = COUNT;

COUNT = COUNT + 1;

at the beginning of procedure SEARCH. Then the vertices of the forest would be labeled $1, 2, \ldots,$ up to the number of vertices in the forest.

The complexity of the algorithm is $O(|V| + |E|)$.

April 14, 2005

15 Biconnectivity

Recall that a graph is connected if for any pair of vertices x and y there is an x-y walk. A component of a graph is a maximal subgraph that is connected. Suppose $G = (V, E)$ has r components $G_i = (V_i, V_i)$ for $1 \leq i \leq r$. It is the case that $V_1, V_2, \ldots, V_r$ is a partition of V; and $E_i, E_2, \ldots, E_r$ is a partition of E. In fact, each G_i is the subgraph induced by V_i. A graph is disconnected if it is not connected.

In a graph G, a *cut-vertex* is a vertex v for which $G - v$ has more components than G. Or equivalently, there are two vertices x and y differ from v such that

there is an x-y walk and every x-y walk contains v.

For the case when G is connected, v is a cut-vertex if and only if $G - v$ is disconnected. A graph is 2-*connected* if it has at least two vertices, is connected and has no cut-vertex. A 2-*connected component* is a maximal 2-connected subgraph. Suppose $G = (V, E)$ is a graph without isolated vertices. Also suppose G has r 2-connected components $G_i = (V_i, V_i)$ for $1 \leq i \leq r$. We have the following propositions.

(1) $|V_i \cap V_j| \leq 1$ for any $i \neq j$.

(2) A vertex v is a cut-vertex of G if and only if $\{v\} = V_i \cap V_j$ for some $i \neq j$.

(3)$E_i, E_2, \ldots, E_r$ is a partition of E.

(4) Each G_i is the subgraph induced by V_i.

Homework 18 *Prove the four properties above for 2-connected components.*

Lemma 30 *Suppose $T = (V, T)$ is a depth-first spanning tree of a connected graph $G = (V, E)$. Vertex v is a cut-vertex if and only if either*

1. *v is a root and v has more than one child, or*

2. *v is not a root, and for some child s of v there is no back edge between any descendant of s (including s itself) and a proper ancestor of v.*

Proof. It is easy to show that the root is a cut-vertex if and only if it has more than one child.

Suppose condition 2 is true. Let p be the parent of v. By Lemma 29 each back edge goes from a vertex to an ancestor of the vertex. Thus any back edge from a descendant v of s goes to an ancestor of v. Hence it goes either to v or to a descendant of s. Thus every path from s to p contains v, implying that v is a cut-vertex.

To prove the converse, suppose that v is a cut-vertex but not the root. Let x and y be distinct vertices other than v such that every path in G between x and y contains v. At least one of x and y, say x, is a proper descendant of v in S, else there is a path in G between x and y using edges in T avoiding v. Let s be the child of v such that x is a descendant of s and a proper ancestor w of v, in which case condition 2 is immediately true, or there is such an edge vw. In the latter situation we must consider two cases.

CASE 1. Suppose y is not a descendant of v. Then there is a path from x to v to w to y that avoids v, a contradiction.

CASE 2. Suppose y is a descendant of v. Surely y is not a descendant of s, else there is a path from x to y that avoids v. Let s' be the child of v such that y is a descendant of s'. Either there is no back edge between a descendant v' of s' and a proper ancestor w' of v in which case condition 2 is immediately true, or there is such an edge $v'w'$. In the latter case there is a path from x to v to w to w' to v' to y that avoids v, a contradiction. We conclude that condition 2 is true. □

Midterm exam on April 19, 2005 | **April 21, 2005**

Let T and B be the sets of tree and back edges produced by a depth-first search of a connected graph $G = (V, E)$. We assume the vertices in V are namely by their depth-first search numbers. For each v in V, we define

$$\text{LOW}[w] = \text{MIN}\ (\{v\} \cup \{w : xw \in B \text{ for some descendant } x \text{ of } v\}).$$

The preorder numbering implies that if x is a descendant of v and xw is a back edge such that $w < v$, then w is a proper ancestor of v. Thus by Lemma 30, if vertex v is not the root, then v is a cut-vertex if and only if v has a child s such that $\text{LOW}[s] \geq v$.

We can embed into the procedure SEARCH a calculation to determine the LOW value of each vertex if we rewrite the above definition to express LOW[v] in the children of v.

Specifically, LOW[v] can be computed by determining the minimum value of those vertices w such that wither

1. $w = v$, or
2. $w =$ LOW[s] and s is a child of v, or
3. vw is a back edge in B.

The minimum value of w can be determine once $N(v)$ is exhausted. Thus the above definition is equivalent to

LOW[v] = MIN($\{v\} \cup \{$LOW[s]: s is a child of $v\} \cup \{w : vw \in B\}$).

We have incorporated both the renaming of the vertices by first visit and the computation of LOW into the revised version of SEARCH shown below. The algorithm for a connected graph is as follows.

Algorithm 2-CONNECTED-COMPONENTS.

```
1.  Initially set T to ∅, STACK to ∅, COUNT to 1.
    Mark each vertex in V as being "new".
2.  while there is a vertex v marked "new" do SEARCHB(v).

    procedure SEARCHB(v) ;
3.      mark v "old" ;
4.      DFSNUMBER[v] = COUNT ;
5.      COUNT = COUNT + 1 ;
6.      LOW[v] = DFSNUMBER[v] ;
7.      push v into STACK ;
8.      for each vertex w ∈ N(v) do
9.          if w is marked "new" then
10.         {   add vw to T ;
11.             PARENT[w] = v ;
12.             SEARCHB(w)
13.             if LOW[w] ≥ DFSNUMBER[v]
14.             then pop elements up to w from STACK together
                        with v to form a 2-connected component ;
15.             LOW[v] = MIN(LOW[v], LOW[w])
16.         }
17.         else if w is not PARENT[v] then
18.             LOW[v] = MIN(LOW[v], DFSNUMBER[w]).
```

April 26, 2005

16 Strong connected components

We now give an application of depth-first search in digraphs.

A digraph G is *strongly connected* if for every two vertices x and y in G there is an x-y dipath. A *strongly connected component* of a digraph is a maximal strongly connected subdigraph. Suppose digraph $G = (V, E)$ has exactly r strongly connected components $G_i = (V_i, E_i)$ where $1 \leq i \leq r$. The follows properties hold.

(i) $V_1, V_2, \ldots, V_r$ is a partition of V.

(ii) $E_i \cap E_j = \emptyset$ for $i \neq j$. (However, $E_1, E_2, \ldots, E_r$ is not necessary a partition of E.)

(iii) G_i is induced by V_i.

(iv) Let $G^* = (V^*, E^*)$ be the digraph with $V^* = \{V_1, V_2, \ldots, V_r\}$ and $E^* = \{V_iV_j : \text{there is an edge of } G \text{ from } V_i \text{ to } V_j\}$. Then G^* is acyclic.

An alternative way to describe strongly connected components of a digraph $G = (V, E)$ is as follows. Define a relation $\sim$ on V as $x \sim y$ if and only if there is a diwalk from x to y and a diwalk from y to x. It is easy to see that this is an equivalent relation and hence it partitions V into equivalent classes $V_1, V_2, \ldots, V_r$, which induce the strongly connected components of G.

If we apply the depth-first search to a digraph, four kind of edges will produced.

1. *Tree edges*, which are edges leading to new vertices during the search.
2. *Forward edge*, which go from ancestors to proper descendants but are not tree edges.
3. *Back edge*, which go from descendants to ancestors (possibly from a vertex to itself).
4. *Cross edges*, which go between vertices that are neither ancestors nor descendants of one another.

Lemma 31 *If vw is a cross edge, then $v > w$, i.e., cross edges go from right to left.*

$$\text{LOWLINK}[v] = \text{MIN}(\{v\} \cup \{w : \begin{array}{l}\text{there is a cross or back edge from a descendant}\\ \text{of } v \text{ to } w, \text{ and the root of the strongly connected}\\ \text{component containing } w \text{ is an ancestor of } v\}).\end{array}$$

Lemma 32 *A vertex is the root of a strongly connected component of G if and only if* $\text{LOWLINK}[v] = v$.

Algorithm STRONGLY-CONNECTED-COMPONENTS.

1. Initially set T to $\emptyset$, STACK to $\emptyset$, COUNT to 1. Mark each vertex in V as being "new".
2. **while** there is a vertex marked "new" **do** SEARCHC(v).

```
    procedure SEARCHC(v) ;
 3.     mark v "old" ;
 4.     DFSNUMBER[v] = COUNT ;
 5.     COUNT = COUNT + 1 ;
 6.     LOWLINK[v] = DFSNUMBER[v] ;
 7.     push v into STACK ;
 8.     for each vertex w ∈ N(v) do
 9.        if w is marked "new" then
10.        {  add vw to T ;
11.           SEARCHC(w)
12.           LOWLINK[v] = MIN(LOWLINK[v], LOWLINK[w])
13.        }
14.        else if DFSNUMBER[w] < DFSNUMBER[v] and w is on STACK then
15.           LOWLINK[v] = MIN(LOWLINK[v], DFSNUMBER[w])
16.     if LOWLINK[v] = DFSNUMBER[v] then
17.           pop elements up to v from STACK form a strongly connected component .
```

Homework 19 *Use depth-frist search to help design efficient algorithms to do the following.*

(a) *Test whether a graph is acyclic.*

(b) *Find an order for the vertices of an acyclic digraph such that $v < w$ if there is a dipath from v to w of length greater than zero.*

(c) *Determine whether the edges of a connected graph can be directed to produce a strongly connected digraph.* [*Hint: Show that this can be done if and only if removing any edge from G leaves a connected graph.*]

April 28, 2005

17 Turing machines

We need a more precise model for an algorithm as follows.

A *k-tape Turing machine* is a seven-tuple $M = (Q, T, I, \delta, \mathrm{b}, q_9, q_f)$, where:

1. Q is the set of *states*.
2. T is the set of *tape symbols*.
3. I is the set of *input symbols*, $I \subseteq T$.
4. b, in $T - I$, is the *blank*.
5. q_0 is the *initial state*.
6. q_f is the *final* (or *accepting*) *state*.

7. δ, the *next-move function*, map a subset of $Q \times T^k$ to $Q \times (T \times \{\mathrm{L}, \mathrm{R}, \mathrm{S}\})^k$. That is, for some $(k+1)$-tuples consisting of a state and k tape symbols, it gives a new state and k pairs, each pair consisting of a new tape symbol and a direction for the tape head. Suppose $\delta(q, a_1, a_2, \ldots, a_k) = (q', (a'_1, d_1), (a'_2, d_2), \ldots, (a'_k, d_k))$, and the Turing machine is in state q with ith tape head scanning tape symbol a_i for $1 \leq i \leq k$. Then in one move the Turing machine enters state q', changes symbol a_i to a'_i, and moves the ith tape head in the direction d_i for $1 \leq i \leq k$.

The string of input symbols is *accepted* if the Turing machine, started in the designed initial state, with all tape heads at the left ends of their tapes, makes a sequence of moves in which it eventually enters the accepting state. The *language accepted* by the Turing machine is the set of strings of input symbols so accepted.

Example. The two-tape Turing machine below recognizes palindromes on the alphabet $\{0, 1\}$. Recall that a palindrome is a string which reads the same backwards as forwards, e.g., 0100010.

Current state	Symbol on: Tape 1	Tape 2	(New symbol, head move) Tape 1	Tape 2	New state	Comments
q_0	0	b	0,S	X,R	q_1	If input is nonempty, print X on
	1	b	1,S	X,R	q_1	tape 2 and move head right; go to
	b	b	b,S	b,S	q_5	state q_1. Otherwise, go to state q_5.
q_1	0	b	0,R	0,R	q_1	Stay in state q_1 copying tape 1 onto
	1	b	1,R	1,R	q_1	tape 2 until b is reached on tape 1.
	b	b	b,S	b,L	q_2	Then go to state q_2.
q_2	b	0	b,S	0,L	q_2	Keep tape 1's head fixed and move
	b	1	b,S	1,L	q_2	tape 2's left until X is reached.
	b	X	b,L	X,R	q_3	Then go to state q_3.
q_3	0	0	0,S	0,R	q_4	Control alternates between states q_3
	1	1	1,S	1,R	q_4	and q_4. In q_3 compare the symbols
q_4	0	0	0,L	0,S	q_3	on two tapes, move tape 2's head
	0	1	0,L	1,S	q_3	right, and go to q_4. In q_4 go to q_5
	1	0	1,L	0,S	q_3	and accept if head has reached b on
	1	1	1,L	1,S	q_3	tape 2. Otherwise move tape 1's
	0	b	0,S	b,S	q_5	head left go back to q_3. The alter-
	1	b	1,S	b,S	q_5	nation q_3, q_4 prevents the input head
						from falling off the left end of tape 1.
q_5						Accept

The activity of a Turing machine can be described by means of "instantaneous description." An *instantaneous description* (ID) of a k-tape Turing machine M is a k-tuple $(\alpha_1, \alpha_2, \ldots, \alpha_k)$ where each α_i is a string of the form xqy such that $xy$is the string on the ith tape of M (with trailing blanks omitted) and q is the current state of M. The symbol immediately to the right of the ith q is the symbol being scanned on the ith tape.

If instantaneous description D_1 becomes instantaneous description D_2 after one move of the Turing machine M, then we write $D_1 \vdash_M D_2$ (read $\vdash$ as "goes to"). If $D_1 \vdash_M D_2 \vdash_M \ldots \vdash_M D_n$ for some $n \geq 2$, then we write $D_1 \vdash_M^+ D_n$. If either $D = D'$ or $D \vdash_M^+ D'$, then we write $D \vdash_M^* D'$. We demon the input string 010, which is accepted, as follows.

$$\begin{aligned}
(q_0010, q_0) &\vdash (q_1010, Xq_1) \\
&\vdash (0q_110, X0q_1) \\
&\vdash (01q_10, X01q_1) \\
&\vdash (010q_1, X010q_1) \\
&\vdash (010q_2, X01q_20) \\
&\vdash (010q_2, X0q_210) \\
&\vdash (010q_2, Xq_2010) \\
&\vdash (010q_2, q_2X010) \\
&\vdash (01q_30, Xq_3010) \\
&\vdash (01q_40, X0q_410) \\
&\vdash (0q_310, X0q_310) \\
&\vdash (0q_410, X01q_40) \\
&\vdash (q_3010, X01q_30) \\
&\vdash (q_4010, X010q_4) \\
&\vdash (q_5010, X010q_5)
\end{aligned}$$

Homework 20 *Design a Turing machine to accept the strings of the form*

$$10^a10^b10^c$$

such that a, b, c are nonnegative integers with $a + b = c$.

May 3, 2005

A *k-tape nondeterministic Truing machine* is a seven-tuple $M = (Q, T, I, \delta, \mathrm{b}, q_9, q_f)$, where all components have the same meaning as for the ordinary (deterministic) Turing machine, except that here the next-move function is a mapping from $Q \times T^k$ to subsets of $Q \times (T \times \{\mathrm{L}, \mathrm{R}, \mathrm{S}\})^k$. That is, given a state and list of k tape symbols, δ returns a finite set of choices of next move; each choice is a new state, with k new tape symbols and k moves of the tape heads. Note that the NDTM M may choose any of these moves, but it cannot choose a next state from one and new tape symbols from another, or make any other combination of moves.

Example. Here is a 3-tape NDTM to accept the strings of the form

$$10^{i_1}10^{i_2}\ldots10^{i_k}$$

such that there is some $I \subseteq \{1, 2, \ldots, k\}$ for which $\sum_{i \in I} i_j = \sum_{j \notin I} i_j$.

State	Current symbol Tape 1	Tape 2	Tape 3	(New symbol, head move) Tape 1	Tape 2	Tape 3	New state	Comments
q_0	1	b	b	1,S	\$,R	\$,R	q_1	Mark left ends of tapes 2 and 3 with \$, then go to state q_1.
q_1	1 1	b b	b b	1,R 1,R	b,S b,S	b,S b,S	q_2 q_3	Here we choose whether to write the next block on tape 2 (q_2) or tape 3 (q_3).
q_2	0 1 b	b b b	b b b	0,R 1,S b,S	0,R b,S b,L	b,S b,S b,L	q_2 q_1 q_4	Copy the block of 0's onto tape 2, then return to state q_1 when 1 is reached on tape 1. If b is reached on tape 1, instead go to state q_4 to compare the lengths of tapes 2 and 3.
q_3	0 1 b	b b b	b b b	0,R 1,S b,S	b,R b,S b,L	0,S b,S b,L	q_3 q_1 q_4	The same as for state q_2, but write on tape 3.
q_4	b b	0 \$	0 \$	b,S b,S	0,L \$,S	0,L \$,S	q_4 q_5	Compare the length of tapes 2 and 3.
q_5								Accept

$$
\begin{aligned}
&(q_0 1010010, q_0, q_0)\\
\vdash\ &(q_1 1010010, \$q_1, \$q_1)\\
\vdash\ &(1q_2 010010, \$q_2, \$q_2)\\
\vdash\ &(10q_2 10010, \$0q_2, \$q_2)\\
\vdash\ &(10q_1 10010, \$0q_1, \$q_1)\\
\vdash\ &(101q_3 0010, \$0q_3, \$q_3)\\
\vdash\ &(1010q_3 010, \$0q_3, \$0q_3)\\
\vdash\ &(10100q_3 10, \$0q_3, \$00q_3)\\
\vdash\ &(10100q_1 10, \$0q_1, \$00q_1)\\
\vdash\ &(101001q_2 0, \$0q_2, \$00q_2)\\
\vdash\ &(1010010q_2, \$00q_2, \$00q_2)\\
\vdash\ &(1010010q_4, \$0q_4 0, \$0q_4 0)\\
\vdash\ &(1010010q_4, \$q_4 00, \$q_4 00)\\
\vdash\ &(1010010q_4, q_4\$00, q_4\$00)\\
\vdash\ &(1010010q_5, q_5\$00, q_5\$00)\\
&\text{Accept}
\end{aligned}
$$

$$
\begin{aligned}
&(q_0 1010010, q_0, q_0)\\
\vdash\ &(q_1 1010010, \$q_1, \$q_1)\\
\vdash\ &(1q_3 010010, \$q_3, \$q_3)\\
\vdash\ &(10q_3 10010, \$q_3, \$0q_3)\\
\vdash\ &(10q_1 10010, \$q_1, \$0q_1)\\
\vdash\ &(101q_3 0010, \$q_3, \$0q_3)\\
\vdash\ &(1010q_3 010, \$q_3, \$00q_3)\\
\vdash\ &(10100q_3 10, \$q_3, \$000q_3)\\
\vdash\ &(10100q_1 10, \$q_1, \$000q_1)\\
\vdash\ &(101001q_3 0, \$q_3, \$000q_3)\\
\vdash\ &(1010010q_3, \$q_3, \$0000q_3)\\
\vdash\ &(1010010q_4, q_3\$, \$000q_3 0)\\
&\text{Halt, no next ID}
\end{aligned}
$$

Homework 21 *Design a NDTM to accept the binary strings*

$$10^{i_1} 10^{i_2} \dots 10^{i_k}$$

such that $i_r = i_s$ *for some* $1 \leq r < s \leq k$.

We define $\mathcal{P}$-TIME (respectively, $\mathcal{NP}$-TIME)to be the set of all languages which can be accepted by DTS's (respectively, NDT's) of polynomial time complexity. That is,

$$\begin{aligned}\mathcal{P}\text{-TIME} = \{L : &\text{ there is a DTM } M \text{ and a polynomial } p(n) \text{ such that} \\ &M \text{ is of time complexity } p(n) \text{ and } L(M) = L\} \text{ and}\end{aligned}$$

$$\begin{aligned}\mathcal{NP}\text{-TIME} = \{L : &\text{ there is a NDTM } M \text{ and a polynomial } p(n) \text{ such that} \\ &M \text{ is of time complexity } p(n) \text{ and } L(M) = L\}.\end{aligned}$$

We shall frequently abbreviate $\mathcal{P}$-TIME to $\mathcal{P}$ and $\mathcal{NP}$-TIME to $\mathcal{NP}$.

A language L_0 in $\mathcal{NP}$ is *nondeterministic polynomial-time complete* (*NP-complete* for short) if the following condition is satisfied: If we are given a deterministic algorithm of time complexity $T(n) \geq n$ to recognize L_0 then for every language L in $\mathcal{NP}$ we can effectively find a deterministic algorithm of time complexity $T(p_L(n))$, where p_L is a polynomial that depends on L. We say L is reducible to L_0.

We say that a language L is *polynomially transformable* to L_0 if there is a deterministic polynomial-time-bounded Turing machine M which will convert each string w in the alphabet of L into a string w_0 in the alphabet of L_0 such that w is in L if and only if w_0 is in L_0.

One way to prove that a language L_0 is NP-complete is to show that L_0 is in $\mathcal{NP}$ and that every language L in $\mathcal{NP}$ can be polynomially transformed to L_0.

Example. (Problem-language relationship.)

Consider the clique problem for graphs. A *k-clique* in a graph G is a set of k vertices in which each pair of distinct vertices are not adjacent. The *clique problem* is, given a graph G and an integer k, does G contain a k-clique?

The instance of the clique problem with the graph $G = (V, E)$, where $V = \{1, 2, 3, 4, 5\}$ and $E = \{(1, 2), (1, 4), (2, 3), (2, 4), (3, 4)(3, 5), (4, 5)\}$ and $k = 3$ could be encoded by the string:

$$3(1, 2)(1, 4)(2, 3)(2, 4)(3, 4)(3, 5)(4, 5).$$

The first integer represent the value of k. Then follow those pairs of vertices connected by edges.

The language L representing the clique problem is the set of strings of the form

$$k(i_1, j_1)(i_2, j_2) \ldots (i_m, j_m).$$

A Boolean expression f is *satisfiable* if we can assign 0's and 1's to the variables giving the expression the value 1. For instance, in $f = (x + \overline{y} + z)(\overline{x} + y + z)$, we may assign x with 1, y with 0 and z with 1 to get f 1.

In a graph $G = (V, E)$, a *vertex cover* is a subset $S \subseteq V$ such that each edge is incident to some vertex in S. A *Hamiltonian circuit* is a cycle containing every vertex of V. G is *k-colorable* if we can assign $1, 2, \ldots, k$ to the vertices of G such that adjacent vertices get different colors.

In a digraph $G = (V, E)$, a *feedback vertex set* is a subset $S \subseteq V$ such that every cycle contains a vertex in S. A *feedback edge set* is a subset $F \subseteq E$ such that every cycle contains an edge in F. A *directed Hamilton circuit* is a cycle containing every vertex of V.

Theorem 33 *The following problems are in $\mathcal{NP}$.*

1. (Satisfiability.) *Is a Boolean expression satisfiable?*
2. (Clique.) *Does a graph have a vertex cover of size k?*
3. (Vertex cover.) *Does a graph have a vertex cover of size k?*
4. (Hamilton circuit.) *Does a graph have a Hamilton circuit?*
5. (Colorability.) *Is a graph k-colorable?*
6. (Feedback vertex set.) *Does a digraph have a feedback vertex set with k members?*
7. (Feedback edge set.) *Does a digraph have a feedback edge set with k members?*
8. (Directed Hamilton circuit.) *Does a digraph have a directed Hamilton circuit?*
9. (Set cover.) *Given a family of sets $S_1, S_2, \ldots, S_n$ does there exist a subfamily of k sets $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$ such that*
$$\cup_{j=1}^{k} S_{i_j} = \cup_{j=1}^{n} S_j?$$

Theorem 34 *The problem of determining whether a Boolean expression is satisfiable is NP-complete.*

Proof. We shall prove that every language L in $\mathcal{NP}$ is polynomially transformable to the satisfiability problem. Let M be the NDTM of polynomial-time complexity $p(n)$ that accept L, and w an input to M. Suppose M has a single tape, has staes $q_1, q_2, \ldots, q_s$, and tape symbols $X_1, X_2, \ldots, X_m$.

It is possible to construct a Boolean expression f, whose size is polynomial to the input length, such that f is satisfiable if and only if M accept w. □

May 5, 2005

Conjunctive normal form (CNF): Suppose there are n Boolean variables $v_1, v_2, \ldots, v_n$. A *literal* is either v_i or $\overline{v}_i$. A Boolean expression in CNF is one of the form

$$(x_{1,1} + x_{1,2} + \ldots + x_{1,k_1})(x_{2,1} + x_{2,2} + \ldots + x_{2,k_2}) \ldots (x_{m,1} + x_{m,2} + \ldots + x_{m,k_m}),$$

where all $x_{i,j}$ are literals.

Corollary 35 *The satisfiability problem for Boolean expressions in CNF is NP-complete.*

Proof. Use rules: (1) $p \to q$ is the same as $\overline{p} + q$; (2) $\overline{p + q}$ is the same as $\overline{p}\ \overline{q}$; (3) $\overline{pq}$ is the same as $\overline{p} + \overline{q}$; (4) $p + (qr)$ is the same as $(p + q)(p + r)$. □

Theorem 36 *3-satisfiability is NP-complete.*

Proof. Replace (x_1) by $(x_1+x_1+x_1)$; (x_1+x_2) by $(x_1+x_2+x_2)$; and $(x_1+x_2+\ldots x_k)$ with $k \geq 4$ by $(x_1+x_2+y_1)(x_3+\overline{y}_1+y_2)(x_4+\overline{y}_2+y_3)\ldots(x_{k-2}+\overline{y}_{k-4}+y_{k-3})(x_{k-1}+x_k+\overline{y}_{k-3})$ for new variables $y_1, y_2, \ldots, y_{k-3}$. (Notice that this follows from that $g+h$ is equivalent to $(g+y)(h+\overline{y})$.) □

Theorem 37 *CNF-satisfiability is polynomially transformable to the clique problem. Therefore, the clique problem is NP-complete.*

Proof. Let $F = F_1F_2\ldots F_q$ be an expression in CNF, where the F_i's are the factors; each F_i is of the form $(x_{i,1}+x_{i,2}+\ldots+x_{i,k_i})$, where $x_{i,j}$ is a literal. We shall construct a graph $G=(V,E)$ whose vertices are pairs of integers $[i,j]$ for $1 \leq i \leq q$ and $1 \leq j \leq k_i$. The first component of the pair represents a factor, and the second a literal within the factor. Thus each vertex of the graph correspondents to a particular literal of a particular factor in a natural way.

The edges of G are those pairs $([i,j],[k,l])$ such that $i \neq k$ and $x_{i,j} \neq \overline{x}_{k,l}$. Intuitively, $[i,j]$ and $[k,l]$ are adjacent in G if they correspond to different factors and it is possible to assign values to the variable in literals $x_{i,j}$ and $x_{k,l}$ in such a way that both literals have value 1. That is either $x_{i,j} = x_{k,l}$, or $x_{i,j}$ and $x_{k,l}$ are complemented or uncomplemented versions of different variables.

We claim that F is satisfiable if and only if G has a clique of size q.

Suppose F is satisfiable, i.e. we may assign 0's and 1's to the variables such that for each i there is at least one x_{i,m_i} is 1. Let $S = \{[i,m_i] : 1 \leq i \leq q$. Then S is a clique for otherwise there exist $i \neq j$ such that there is no edge between $[i,m_i]$ and $[j,m_j]$, implying $x_{i,m_i} = \overline{x}_{j,m_j}$ by the definition of G. But this is impossible since $x_{i,m_i} = x_{j,m_j} = 1$ by the way the x_{i,m_i}'s were selected.

On the other hand, suppose G has a clique S of size q. Each vertex in the clique must have a distinct first component, say $S = \{[i,m_i] : 1 \leq i \leq q\}$. Let $S_1 = \{y : x_{i,m_i} = y$, where $1 \leq i \leq q$ and y is a variable$\}$; and $S_2 = \{y : x_{i,m_i} = \overline{y}$, where $1 \leq i \leq q$ and y is a variable$\}$. Then $S_1 \cap S_2 = \emptyset$. So we may assign variables of S_1 to 1 and those in S_2 to 0 to make the value of each F_i to 1. □

Homework 22 (1) *Polynomially transform the clique problem to the vertex cover problem.*

(2) *Polynomially transform the vertex cover problem to the feedback edge problem.*

(3) *Polynomially transform the vertex cover problem to the directed Hamilton circuit problem.*

(4) *Polynomially transform the vertex cover problem to the set cover problem.*

(5) *Polynomially transform the 3-satisfiability problem to the colorability problem.*

(6) *Polynomially transform the colorability problem to the exact cover problem.*

Homework 23 *Design an efficient algorithm for the 2-satisfiability problem.*

18 Atfer May 3, 2005

From now on, there is no detail lecture note. Please read Evan's book [6]. For some part of the book, see the web. Other useful books are [7, 10]. I will only put some homework here for your refeences.

Homework 24 *Suppose $G = (V, E)$ is a digraph in which s and t are two specified vertices, and each edge e has a non-nenative capacity $c(e)$. If there are r dipaths from s to t: $P_1, P_2, \ldots, P_r$. Suppose $a_1, a_2, \ldots, a_r$ are nonnegative numbers. Let*

$$f_{a_1,\ldots,a_r}(e) = \sum\{a_i : 1 \le i \le r, e \in E(P_i)\}$$

for any edge e in G.

(1) *Prove that $f_{a_1,\ldots,a_r}$ is an s-t flow if and only if $f_{a_1,\ldots,a_r}(e) \le c(e)$ for all $e \in E$. And in this case,* $\text{value}(f_{a_1,\ldots,a_r}) = \sum_{i=1}^{r} a_i$.

(2) *From* (1) *it is then clear that* $\max\{\text{value}(f) : f \text{ is an } s\text{-}t \text{ flow}\} \ge \max\{\text{value}(f_{a_1,\ldots,a_r}) : a_1, a_2, \ldots, a_r \text{ are nonnegative numbers and } f_{a_1,\ldots,a_r} \text{ is an } s\text{-}t \text{ flow}\}$. *Prove that this is indeed an equality.*

Homework 25 *Problem* 6.1 *in Page* 142 *of Even's book.*

Homework 26 *Problem* 6.3 *in Page* 143 *of Even's book.*

Homework 27 *Problem* 6.13 *in Page* 146 *of Even's book.*

On May 31 and June 2, I will talk on the $L(2,1)$-labelings on graphs, see the papers:

(1) J. R. Griggs and R. K. Yeh, Labeling graphs with a condition at distance 2, SIAM J. Discrete Math. 5 (1992) 586-595.

(2) G. J. Chang and D. Kuo, The $L(2,1)$-labeling problem on graphs, SIAM J. Discrete Math. 9 (1996) 309-316.

(3) G. J. Chang and S.-C. Liaw, The $L(2,1)$-labeling problem on ditrees, Ars Combin. 66 (2003) 23-31.

(4) G. J. Chang, J.-J. Chen, D. Kuo, and S.-C. Liaw, Distance-two labelings of digraphs, http://tw.arxiv.org/abs/math/0407168.

Homework 28 *Determine $\lambda(C_n)$ for $n \ge 3$. Prove your answer.*

Homework 29 *Prove that for a graph G of n vertices, $\lambda(G) \le n - 1$ if and only if the complement of G has a Hamiltonian cycle.*

Homework 30 *Prove that for a ditree T, we have $\vec{\lambda}(T) \le 4$.*

References

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1974.

[2] R. K. Ahuja, T. L. Magnanti andJ. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall International, London, 1993.

[3] G. Chartrand and O. R. Oellermann, *Applied and Algorithmic Graph Theory*, McGraw-Hill, New York, 1993.

[4] W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, A. Schrijver, *Combinatorial Optimization*, John Wiley & Sons, Inc., New York, 1998.

[5] T. H. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction to Algorithm*, Second Edition, The MIT Press, Cambridge, 2002.

[6] S. Even, *Graph Algorithms*, 1979.

[7] L. R. Ford and D. R. Fulkerson, *Flows in Networks,* Princeton University Press, Princeton, NewJersey, 1962.

[8] A. Gibbons, *Algorithmic Graph Theory*, Cambridge University Press, 1985.

[9] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

[10] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids,* Hold, Rinehart and Winston, New York, 1976.

[11] J. A. McHugh, *Algorithmic Graph Theory*, Prentice-Hall, London, 1990.

[12] K. Thulasiraman and M. N. S. Swamy, *Graphs: Theory and Algorithms*, John Wiley & Sons, Inc., New York, 1992.

[13] D. J. A. Welsh, *Matroid Theory*, Academic Press, London, 1976.

[14] H. Whitney, "A set of topological invariants for graphs," *Amer. J. Math.* 55 (1933) 221-235.