Scribbles on some topics in algorithms

by Ranveer[1]

Contents

Notations

Throughout the text, the matrices (vectors) without subscript are the matrices (vectors) of a suitable order. By a vector x we mean a column vector.

1. O : All-zero Matrix
2. J : All-one Matrix
3. $\mathbf{0}$: All-zero vector
4. $\mathbf{1}$: All-one vector
5. I : Identity matrix
6. A^T : Transpose of a matrix A
7. $|S|$: Cardinality of set S, that is, the number of elements in S.
8. $S \setminus T$: The set resulting after removing all the elements of T from S.

Chapter 1

Counting the number of spanning trees

Ranveer, CSE, IIT Indore

In an undirected graph G a *spanning tree* is a subgraph that is connected, acyclic, and covers all the vertices of G. See graph in Figure 5.5(a), all its spanning trees are shown in Figure 1.2 (in green). Matrix-Tree Theorem gives the number of spanning-tree in an undirected graph in a polynomial time, $O(n^{2.37})$, where n is the number of vertices in G. Before we state and prove the theorem we need some preliminaries and notations.

1.1 Incidence matrix and Laplacian matrix

Let G be an undirected graph with vertex-set $V(G) = \{1, \ldots, n\}$ and edge-set $E(G) = \{e_1, \ldots, e_m\}$. To each edge of G assign an orientation (direction), which is arbitrary. The *incidence matrix* of G, denoted by $Q(G)$ (often just Q), is the $n \times m$ matrix defined as follows. The rows and the columns of Q are indexed by $1, \ldots, n$

Figure 1.1: (a) A spanning tree is shown in green. (b) An arbitrary orientation and labelling of edges.

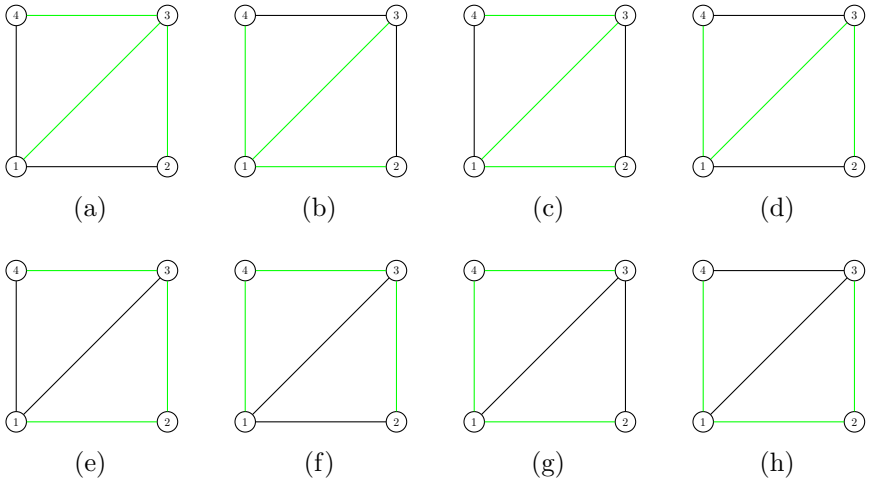


Figure 1.2: All the spanning trees (in green) of the graph in Figure 5.5 (a).

and $1, \ldots, m$, respectively. The (i, j)-entry of Q is 0 if edge e_j is not incident on vertex i, otherwise it is 1 or -1 according as e_j originates or terminates at i, respectively. The matrix $L = QQ^T$ is called the *Laplacian matrix* of G.

Example 1.1.1. *The incidence matrix and Laplacian matrix for the graph in* Figure 5.5 (a) *corresponding to a orientation of edges in* Figure 5.5 (b) *are as follows.*

$$Q = \begin{bmatrix} 1 & 0 & 0 & -1 & 1 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & -1 \\ -1 & 0 & -1 & 0 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 2 & -1 & 0 \\ -1 & -1 & 3 & -1 \\ -1 & 0 & -1 & 2 \end{bmatrix}$$

For any undirected graph check:

1. Adding all the rows of Q gives a zero row vector. Adding all the rows (columns) of L gives a zero row (column) vector.

2. Relabelling the vertices and edges of G does not change the rank of Q, does not changes the rank, determinant of L.

3. If G has n vertices and $n - 1$ edges, then G is either a tree or it is disconnected, that is, G has more than one connected component.

4. If G is a tree having at least two vertices, then there are at least two pendant vertices (vertices which are incident on exactly one edge).

Some more notations: Let A be an $n \times m$ matrix and $S \subset \{1, \ldots, n\}, T \subset \{1, \ldots, m\}$, the notation $A[S|T]$ denotes the sub-matrix of A whose rows and columns

indices are in S, T, respectively. And the notation $A(S|T)$ denotes the sub-matrix of A whose rows and columns indices are not in S, T, respectively.

Matrix-Tree Theorem: *Any cofactor of L equals the number of spanning trees in G.*

In order to prove the Matrix-Tree Theorem we first need the following two lemmas and Cauchy-Binet Theorem.

Lemma 1.1.2. *All the cofactors of L are equal.*

Proof. As mentioned earlier the sum of all the rows of L gives a zero vector. Consider the sub-matrix $L(1|1)$. Its first row vector is $[L(2,2), L(2,3), \ldots, L(2,n)]$. Now consider the sub-matrix $L(2|1)$. Except its first row add all the other rows to the first one, this gives the row vector $-[L(2,2), L(2,3), \ldots, L(2,n)]$. Hence $\det(L(2|1)) = -\det(L(1|1))$. Similarly, we can prove that $\det(L(i|i)) = (-1)^{j+i} \det(L(j|i))$ for any i, j. Hence all the cofactors are the same. □

Lemma 1.1.3. *Let G be a connected graph on n vertices and m edges. If $T \subset \{1, \ldots, m\}$ with $|T| = n-1$, and let H be the subgraph of G consisting of edges that correspond only to the elements of T, then H forms a spanning tree of G if and only if* $\det Q_G[\{1, \ldots, n-1\}|T] = \pm 1$.

Proof. First, suppose that H is not a spanning tree of G. This is possible in two cases.

1. H is subgraph on less than n vertices. Suppose $|V(H)| = k \leq n-1$. Without loss of generality let vertex $n \notin V(H)$, then the submatrix $Q_G[\{1, \ldots, n-1\}|T]$ is the incidence matrix Q_H with $n-1-k$ zero rows. Thus the column sum of $Q_G[\{1, \ldots, n-1\}|T]$ gives a zero row, hence $\det Q_G[\{1, \ldots, n-1\}|T] = 0$.

2. H is an spanning subgraph on n vertices but not a tree. Note that as there are $n-1$ edges, in this case there has to be at least two components in H (Observation 3). Let C be a connected component of H, and without loss of generality let $n \notin V(C)$, and let $V(C) = \{1, \ldots, k\}, k < n$. The sub-matrix $Q_G[\{1, \ldots, n-1\}|T]$ can be written as

$$Q_G[\{1, \ldots, n-1\}|T] = \begin{bmatrix} Q_C & \mathbf{0} \\ \mathbf{0} & \tilde{Q}_C \end{bmatrix},$$

where Q_C is the incidence matrix of a component C, and $\tilde{Q}_C$ is the incidence matrix of subgraph induced by the edges of H not in C. In $Q_G[\{1, \ldots, n-$

$1\}|T]$ adding the rows $2, \ldots, k$ to the first row results in the zero row. Hence, $\det Q_G[\{1, \ldots, n-1\}|T] = 0$.

Conversely, suppose H is a spanning tree of G on the edges $e_1, \ldots, e_{n-1}$. Note that, H must have at least two vertices with degree 1. Pick one such a vertex, and without loss of generality assume that e_1 is incident on this vertex, now label this vertex as 1. Next, consider the tree on $V(H) \setminus 1$ vertices, and pick a vertex with degree 1 in it, assume that e_2 is incident on this vertex, and label this vertex as 2. Continue such a labeling of vertices for all edges $e_1, \ldots, e_{n-1}$. This relabelling of vertices and edges gives a $n-1$ order square matrix which is a permutation of rows and columns of matrix $Q_G[\{1, \ldots, n-1\}|T]$. Note that this resulting matrix is a lower triangular matrix with diagonal entries ± 1. Hence, $\det Q_G[\{1, \ldots, n-1\}|T] = \pm 1$. □

We are now just one step away from proving the Matrix-Tree Theorem. For the final step we need the following famous theorem known as Cauchy-Binet Theorem.

Theorem 1.1.4 (**Cauchy-Binet Theorem**). *Let A and B be two $n \times m$ and $m \times n$ matrices, respectively, for some positive integers n and m with $n \leq m$. Then*

$$\det AB = \sum_T \det A[\{1, \ldots, n\}|T] \det B[T|\{1, \ldots, n\}],$$

where the summation runs over all subsets T of $\{1, \ldots, m\}$ with $|T| = n$.

Example 1.1.5. *Consider* $A = \begin{bmatrix} 2 & -1 & 3 \\ 0 & -1 & 0 \end{bmatrix}$, $B = \begin{bmatrix} 3 & -1 \\ -1 & 2 \\ -1 & -1 \end{bmatrix}$. $AB = \begin{bmatrix} 4 & -7 \\ 1 & -2 \end{bmatrix}$, $\det AB = -1$. *By Cauchy-Binet Theorem*

$$\begin{aligned} \det AB = & \det \begin{bmatrix} 2 & -1 \\ 0 & -1 \end{bmatrix} \det \begin{bmatrix} 3 & -1 \\ -1 & 2 \end{bmatrix} + \det \begin{bmatrix} 2 & 3 \\ 0 & 0 \end{bmatrix} \det \begin{bmatrix} 3 & -1 \\ -1 & -1 \end{bmatrix} \\ & + \det \begin{bmatrix} -1 & 3 \\ -1 & 0 \end{bmatrix} \det \begin{bmatrix} -1 & 2 \\ -1 & -1 \end{bmatrix} \\ & = -1. \end{aligned}$$

We are now ready to prove the Matrix-Tree Theorem.

Proof of Matrix-Tree Theorem: Without loss of generality we prove that $\det L_G(n|n)$

equals the number of spanning trees in G.

$$\begin{aligned}\det L_G(n|n) &= \det Q_G Q_G^T(n|n) \\ &= \sum_T (\det Q_G[1,\ldots,n-1|T] \det Q_G^T[T|1,\ldots,n-1]) \quad \text{(Cauchy-Binet Theorem)} \\ &= \sum_T (\det Q_G[1,\ldots,n-1|T])^2,\end{aligned}$$

where the summation runs over all the subsets T of $\{1,\ldots,m\}$, with $|T| = n-1$. By Lemma 1.1.3 $\det Q_G[1,\ldots,n-1|T] = \pm 1$ when the edges corresponding to T forms a spanning tree of G. This completes the proof.
For more on Matrix-Tree Theorem see [1, 3].

Exercise 1.1.6. *Find the number of spanning in complete bipartite graph $K_{m,n}$, cycle graph C_n, and complete graph K_n.*

Chapter 2

Calculating determinant in polynomial time (n^3)

Ranveer, CSE, IIT Indore

In schools most likely nobody have asked us to calculate the determinant of a square matrix of order 4 or more. What could have been the possible reason? The reason is the way we use to calculate the determinant. Let us recall it. Let $M = (m_{ij})$ be a square matrix of order n. The determinant of M, denoted $\det M$, follows the recurrence relation

$$\det M = \sum_{j=1}^{n} (-1)^{i+j} m_{ij} \det M(i|j), \tag{2.1}$$

where $M(i|j)$ is the submatrix resulting after deleting the i-th row and j-th column from matrix M. This recursive procedure is known as Laplace expansion. Computationally, for the determinant of M using Laplace expansion, we will first select a row (or column), and then for each of its entries, we need to find the determinant of a square matrix order $n-1$. Suppose it takes $T(n)$ time to compute the determinant of a $n \times n$ order matrix, then we have the following recurrence relation

$$T(n) = nT(n-1), T(1) = 1.$$

Which gives $T(n) = n!$, thus the time complexity is $O(n!)$, that is, the time grows very, very fast!! So using Laplace expansion calculating the determinant of a 100×100 is impossible by any supercomputer in the lifetime of this planet.
But wait!! Try to find the determinant of a 100×100 matrix on your computer using some computing software; most likely, within seconds, it will calculate it. Certainly, it

is not using the Laplace expansion, then what is it? Here is the trick. Every square matrix M can be decomposed as follows.

$$PM = LU, \tag{2.2}$$

where P is a permutation matrix (that is, a matrix that has exactly one entry of 1 in each row and each column and 0s elsewhere). Matrix L is a lower triangular matrix, that is, its (i, j)-th entry is zero if $(i < j)$. Matrix U is an upper triangular matrix, that is, its (i, j)-th entry is zero if $(i > j)$. This decomposition is also known as LUP decomposition. We might have performed this decomposition while inverting some nonsingular matrix. What is more special about this decomposition? Take determinant both the sides; it will be

$$\det P \det M = \det L \det U, \tag{2.3}$$

(why? see Exercise 2.0.1). $\det P$ is either 1 or -1 (why?), and $\det L, \det U$ is the product of diagonal entries of L, U, respectively. So the $\det M$ can be efficiently calculated if the LUP decomposition does not take much time. The good news is that the time complexity of LUP decomposition (and hence determinant, also rank) of a square matrix of order n is the same as the time complexity of the product of two square matrices of order n. This amazing result we won't prove here, it is given in [2](Chap 17). (At present the fastest way to multiply two square matrices of order n takes $O(n^{2.372})$.) So that's how our computer can solve the determinant of 100×100 matrix within seconds (Although most of computing software are using standard algorithm to multiply $O(n^3)$ the matrices).

Exercise 2.0.1. *Let A and B be two square matrices of order n, then prove that* $\det AB = \det A \det B$.

2.1 Computing LUP decomposition

Let us visualise a square matrix $A = (a_{ij})$ of order n as

$$A = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & w^T \\ v & A' \end{bmatrix}.$$

that is, $w = \begin{bmatrix} a_{12} \\ \vdots \\ a_{1n} \end{bmatrix}$, $v = \begin{bmatrix} a_{21} \\ \vdots \\ a_{n1} \end{bmatrix}$, $A' = (a_{ij}), i, j = 2, \ldots, n$. For the time being assume that a_{11} is nonzero. The top leftmost entry of a matrix we will call the pivot, here a_{11} is the pivot. We will first discuss a simple case of LUP decomposition that we call LU decomposition, where $A = LU$.

2.1.1 LU decomposition

We can factor A as

$$A = \begin{bmatrix} a_{11} & w^T \\ v & A' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \frac{v}{a_{11}} & I_{n-1} \end{bmatrix} \begin{bmatrix} a_{11} & w^T \\ 0 & A' - \frac{vw^T}{a_{11}} \end{bmatrix}. \tag{2.4}$$

Suppose

$$A' - \frac{vw^T}{a_{11}} = L'U',$$

for some unit lower triangular matrix L' (diagonal entries are 1), upper triangular matrix U'. Thus

$$A = \begin{bmatrix} 1 & 0 \\ \frac{v}{a_{11}} & I_{n-1} \end{bmatrix} \begin{bmatrix} a_{11} & w^T \\ 0 & L'U' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \frac{v}{a_{11}} & L' \end{bmatrix} \begin{bmatrix} a_{11} & w^T \\ 0 & U' \end{bmatrix}.$$

It is of the form LU. Thus if we know LU decomposition of submatrix $A' - \frac{vw^T}{a_{11}}$, then we can calculate LU decomposition of A. It gives a recursive algorithm to calculate the LU decomposition of A.

A possible disaster

Check that a_{11} must be nonzero. Also the top left most entry of $A' - \frac{vw^T}{a_{11}}$ must be nonzero in order to perform the above decomposition (the top left entry at every recursive step must be nonzero). What could be possible way to get rid of a zero element in the top left entry? We swap the first row of the matrix with a row whose leftmost element is nonzero.

2.1.2 LUP Decomposition

For numerical stability purpose, we select the row whose leftmost element is maximum among the first column elements. Suppose that row is the k-th row. Then, for

some permutation matrix Q.

$$QA = \begin{bmatrix} a_{k1} & w^T \\ v & A' \end{bmatrix}$$

(Note that there has to be at least one nonzero element in the first column of A, otherwise $\det A$ is zero straightforward.) We can write

$$QA = \begin{bmatrix} 1 & 0 \\ \frac{v}{a_{k1}} & I_{n-1} \end{bmatrix} \begin{bmatrix} a_{k1} & w^T \\ 0 & A' - \frac{vw^T}{a_{k1}} \end{bmatrix}.$$

Now suppose, there exists a permutation matrix P', such that

$$P'\left(A' - \frac{vw^T}{a_{k1}}\right) = L'U',$$

for some unit lower triangular matrix L', upper triangular matrix U'. Let

$$P = \begin{bmatrix} 1 & 0 \\ 0 & P' \end{bmatrix} Q.$$

Check that P is also a permutation matrix. Matrix P is the updated permutation matrix for the original matrix A. We get P from matrices Q, that permutation matrix to swaps the first row of A, and $\begin{bmatrix} 1 & 0 \\ 0 & P' \end{bmatrix}$, that swaps the first row of $A' - \frac{vw^T}{a_{k1}}$. We have,

$$\begin{aligned} PA &= \begin{bmatrix} 1 & 0 \\ 0 & P' \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \frac{v}{a_{k1}} & I_{n-1} \end{bmatrix} \begin{bmatrix} a_{k1} & w^T \\ 0 & A' - \frac{vw^T}{a_{k1}} \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ P'\frac{v}{a_{k1}} & P' \end{bmatrix} \begin{bmatrix} a_{k1} & w^T \\ 0 & A' - \frac{vw^T}{a_{k1}} \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ P'\frac{v}{a_{k1}} & I_{n-1} \end{bmatrix} \begin{bmatrix} a_{k1} & w^T \\ 0 & P'\left(A' - \frac{vw^T}{a_{k1}}\right) \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ P'\frac{v}{a_{k1}} & I_{n-1} \end{bmatrix} \begin{bmatrix} a_{k1} & w^T \\ 0 & L'U' \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ P'\frac{v}{a_{k1}} & L' \end{bmatrix} \begin{bmatrix} a_{k1} & w^T \\ 0 & U' \end{bmatrix}. \end{aligned}$$

It is of the form LUP. Thus if we know LUP decomposition of submatrix $A' - \frac{vw^T}{a_{k1}}$, then we can calculate LU decomposition of A. It gives a recursive algorithm to calcu-

late the LU decomposition of A.
We perform the LUP decomposition in $O(n^3)$ by simple recursive procedure. We leave this as an exercise to you.

An example

Let

$$A = \begin{bmatrix} 4 & 0 & 8 \\ 2 & 4 & 2 \\ 5 & 2 & 5 \end{bmatrix}.$$

In the first column 5 is the largest entry, hence swap the third row with the first one. This is equivalent to left multiplying permutaion matrix

$$Q = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

with A, that is,

$$QA = \begin{bmatrix} 5 & 2 & 5 \\ 2 & 4 & 2 \\ 4 & 0 & 8 \end{bmatrix}.$$

Which is seen as

$$PA = \begin{bmatrix} 1 & 0 & 0 \\ 0.4 & 1 & 0 \\ 0.8 & 0 & 1 \end{bmatrix} \begin{bmatrix} 5 & 2 & 5 \\ 0 & 3.2 & 0 \\ 0 & -1.6 & 4 \end{bmatrix}$$

Now focus on the submatrix $A' = \begin{bmatrix} 3.2 & 0 \\ -1.6 & 4 \end{bmatrix}$. In the first column 3.2 is the largest entry so no need to swap the rows. So our permutation matrix $P' = I_2$. And

$$P'A' = \begin{bmatrix} 1 & 0 \\ -0.5 & 1 \end{bmatrix} \begin{bmatrix} 3.2 & 0 \\ 0 & 4 \end{bmatrix}.$$

Next $P = \begin{bmatrix} 1 & 0 \\ 0 & P' \end{bmatrix} Q$. Finally

$$PA = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 4 & 0 & 8 \\ 2 & 4 & 2 \\ 5 & 2 & 5 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0.4 & 1 & 0 \\ 0.8 & -0.5 & 1 \end{bmatrix} \begin{bmatrix} 5 & 2 & 5 \\ 0 & 3.2 & 0 \\ 0 & 0 & 4 \end{bmatrix}.$$

Exercise 2.1.1. *In the form 2.4 prove that if A is nonsingular, then $A' - vw^T/a_{11}$*

must be nonsingular.

Exercise 2.1.2. *A symmetric matrix A is called positive semi-definite if for all nonzero vectors x we have $x^T Ax \geq 0$. Prove that*

1. *LU decomposition of a positive-definite matrix never causes a division by 0. That is, at every iteration the pivot are nonzero.*
2. *any submatrix of order $(n-1) \times (n-1)$ of the Laplacian matrix of a graph on n vertices is always positive semi-definite.*

Exercise 2.1.3. *Find the asymptotic time complexity of the inverse of a square matrix. Establish some dependency of the complexity of matrix inversion with the time complexity of the determinant.*

Chapter 3

A Maximum Bipartite Matching and Number of Perfect Matchings

Ranveer, CSE, IIT Indore

A *matching* M in an undirected graph G is a set of edges such that every vertex of G is incident to at most one edge in M. (In other words it is a set of vertex-disjoint edges.) The size of a matching is the number of edges in that matching. A matching is maximum when it has largest possible size.

Example 3.0.1. *Consider the following bipartite graph* $G = (X \cup Y, E)$ *of girls and boys, where* $X = \{g_1, g_2, g_3, g_4\}$, $Y = \{b_1, b_2, b_3, b_4, b_5\}$. *A matching* M *is shown in green, its size is* 3. $M = \{(g_1, b_4), (g_2, b_1), (g_3, b_3)\}$.

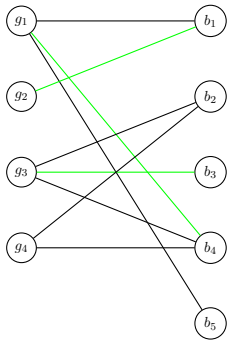


Figure 3.1: $|M| = 3$, is it a maximum matching?

A perfect matching in a graph is a matching that covers every vertex. If a perfect matching does not exists, we are interested to find a maximum matching. Let M be

an arbitrary matching. If M is not maximum, how can we improve it? We will first discuss an algorithm which uses special paths in the graph.

3.0.1 Augmenting path algorithm

An alternating path wrt some matching M is a path that where the edges alternate in M and not in M. For example in Figure 3.1 an alternate path wrt to M is b_5—g_1—b_4—g_3. (If a path consists of just a single edge then it is also an alternating path.)

Finally, an augmenting path is an alternating path that starts and ends on unmatched vertex. For example in Figure 3.1 an augmenting path wrt to M is b_5—g_1—b_4—g_4.

Algorithm 1 Augmenting path algorithm

Input: $G = (X \cup Y, E)$
Output: A maximum matching M
Initialize: $M = \phi$;
while (there is an augmenting path P wrt M)
{
$M = M \oplus P = (M - P) \cup (P - M)$; It is the symmetric difference between M and P.
}
return M;

3.0.2 An illustration

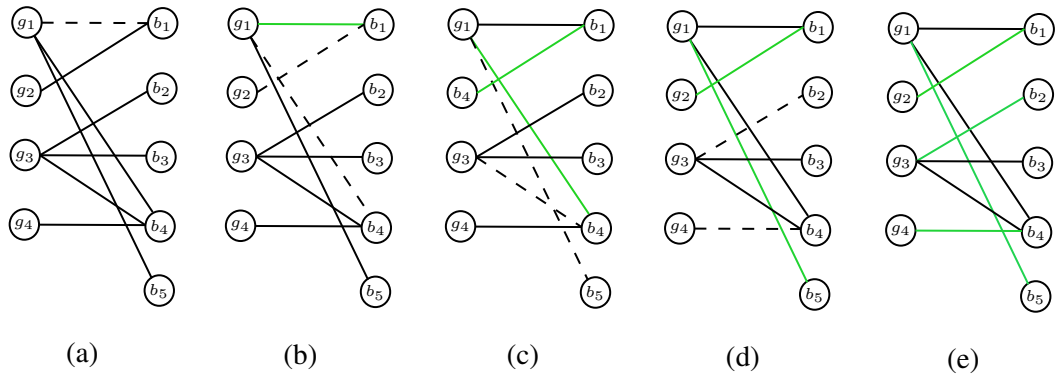


Figure 3.2: Iterations. (In an augmenting path a dashed edge denotes an edge not in matching.)

(a) 1st iteration : $M = \phi$, $|M| = 0$, $P = \{(g_1, b_1)\}$.

(b) 2nd iteration : $M = (M - P) \cup (P - M) = \{(g_1, b_1)\}$, $|M| = 1$.
$P = \{(b_4, g_1), (g_1, b_1), (b_1, g_2)\}$.

(c) 3rd iteration : $M = \{(b_4, g_1), (b_1, g_2)\}$, $|M| = 2$.
$P = \{(b_5, g_1), (g_1, b_4), (b_4, g_3)\}$.

(d) 4th iteration : $M = \{(b_1, g_2), (b_5, g_1), (b_4, g_3)\}$, $|M| = 3$.
$P = \{(b_2, g_3), (g_3, b_4), (b_4, g_4)\}$.

(e) Final iteration : $M = \{(b_1, g_2), (b_5, g_1), (b_2, g_3), (g_4, b_4)\}$, $|M| = 4$.
Now, there is no augmenting path so it is a maximum matching.

Exercise 3.0.2. *After the first iteration in Algorithm 1 is it necessary that an augmenting path must have a matching edge?*

3.0.3 Why the algorithm is correct?

Lemma 3.0.3. *Every component of the symmetric difference of two matchings is a alternating path or an alternating even cycle.*

Example

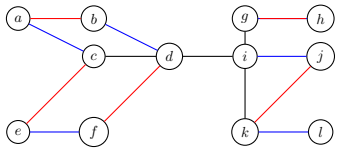


Figure 3.3: Two mathcing M (shown in red), and M' (shown in blue)

$M = \{(a, b), (c, e), (d, f), (g, h), (k, j)\}$
$M' = \{(a, c), (b, d), (e, f), (g, h), (i, j), (k, l)\}$
$(M - M') \cup (M' - M) = \{(a, b), (c, e), (d, f), (k, j), (a, c), (b, d), (e, f), (i, j), (k, l)\}$.

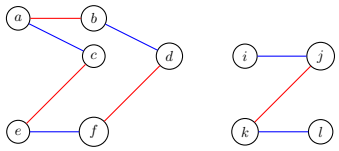


Figure 3.4: The subgraph on the symmetric difference of M and M'

Proof of Lemma 3.0.3
Let M, M' be two matchings. Let $D = (M - M') \cup (M' - M)$. Check that in D a vertex has at most one incident edge from $M - M'$ and at most one incident edge from $M' - M$. Thus the maximum degree of any vertex in D is at most 2. Thus every component of symmetric difference is either a path or a cycle. Furthermore, in every

path or cycle in D edges alternate between $M - M'$ and $M' - M$. Hence any cycle has to be even.

Theorem 3.0.4 (Berge, 1957)**.** *A matching is maximum if and only if there is no augmenting path wrt M.*

Proof. $\implies$ Let M be a maximum matching. Suppose there is an augmenting path P wrt M. Then

$$M' = (M - P) \cup (P - M).$$

But $|M'| = |M| + 1$, so M can not be a maximum matching, which leads to a contradiction.

$\impliedby$ Suppose that there is no augmenting path wrt M and assume that M is not a maximum. Let M' be maximum matching, so $|M'| > |M|$. Let $D = (M - M') \cup (M' - M)$. Check that in D there are more edges from M' than from M. By Lemma 3.0.3 every component of D is either a path or an even cycle. The edges of every path and cycle in D alternate in $M - M'$ and $M' - M$. There must be an alternating path with more edges from M' than M. This path is augmenting path wrt M, which is a contradiction. □

We can generalise the idea in the proof of Theorem 3.0.4 to the following theorem.

Theorem 3.0.5. *Let G be an undirected graph on n vertices, and let M and M' be matchings in G such that $|M'| = |M| + k$, where $k \geq 1$. Then*

1. *there are at least k vertex disjoint augmenting paths in G wrt M.*

2. *at least one of these k augmenting paths is having length at most $\frac{n}{k} - 1$. (that is, at most $\frac{n}{k}$ vertices.)*

Proof.

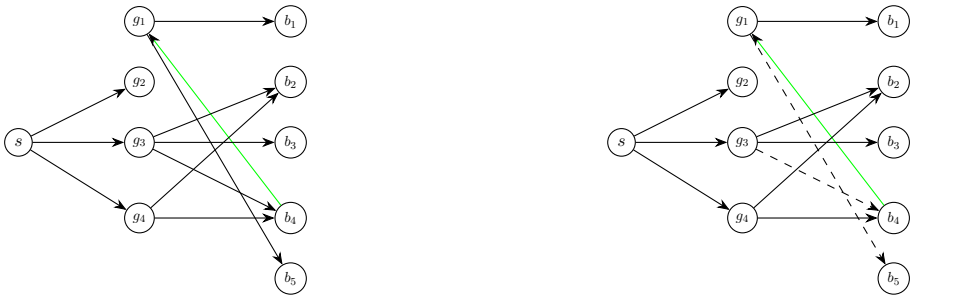
1. By Lemma 3.0.3 $M \oplus M'$ consists of alternating paths or alternating even cycles. Furthermore, $M \oplus M'$ contains exactly k more edges from M' than from M. So there must be at least k vertex disjoint alternating paths which start and end at vertices of M'. These are augmenting paths wrt to M.

2. On contrary assume that all the augmeting paths have more that $\frac{n}{k}$ vertices. But since these are vertex disjoint paths, total vertices in them will exceed n which is a contradiction. So there must be an augmeting path having length at most $\frac{n}{k} - 1$.

□

3.0.4 Implementation and the time complexity analysis

So how to find an augmenting path in a bipartite graph $G = (X \cup Y, E)$ with respect to a matching M? Here is a trick. Convert G in to a directed graph $D(M)$ as follows. To nonmatching edges put direction from X to Y and to matching edges put direction from Y to X. Additionally add an isolated vertex s and add directed edges from s to all nonmatched vertices in X. For example see a $D(M)$ on the left of the following figure.



(a) $D(M)$ (b) An augmenting path using DFS

Start DFS from s. As soon as we hit an unmatched vertex in Y we get a augmented path. Since every augmented path increases a matching size by 1, at most $\frac{n}{2}$ times we have to find an augmenting path. Thus the time complexity is $O(nm)$.

3.1 Hopcroft–Karp algorithm

What if we select multiple augmenting paths instead of just one in augmenting path algorithm? Intuitively it will give a faster way. But we must be careful; the paths should be vertex disjoint. The next algorithm, known as Hopcroft-Karp algorithm is based on this idea. However, wrt to some matching, the augmenting paths it select are not only vertex disjoint but also form a maximal set of shortest paths. Here is the algorithm.

Algorithm 2 Hopcroft–Karp Algorithm

1: $M \leftarrow \emptyset$
2: **repeat**
3: $\quad \mathcal{P} \leftarrow \{P_1, P_2, \ldots, P_k\}$ *a maximal set of vertex-disjoint shortest augmenting paths wrt* M
4: $\quad M \leftarrow M \oplus (P_1 \cup P_2 \cup \cdots \cup P_k)$
5: **until** $\mathcal{P} = \emptyset$

The proof of the correctness of Algorithm 2 is the same as that for augmenting path algorithm. However implementation is more involved. In each phase it uses BFS and DFS to find the set $\mathcal{P}$, a maximal set of vertex-disjoint shortest augmenting paths wrt some matching.

3.1.1 Implementation

Let $G = (U \cup V, E)$ be a given bipartite graph. Assign direction to the edges as follow.

1. All unmatched edges in graph have direction from vertex in set U to vertex in set V.

2. All matched edges in graph have direction from vertex in set V to vertex in set U.

Following (1), initially all edges in graph are unmatched so, they have direction from set U to set V.

3.1.2 BFS Tree Generation

We use multi-source BFS to find the shortest augmenting paths.

1. The source nodes in the BFS tree are the unmatched vertices in U. If there are no unmatched vertices in U or the depth of tree reached is less than or equal to previous depth, the algorithm ends.

2. BFS stops at the level we find an unmatched vertex. This unmatched vertex always belongs to the set V. We store the depth of tree for current iteration.

Note:

- Only the root and leaf nodes in BFS tree can be unmatched vertices.

- Every odd step in the tree will be an unmatched edge as we are moving from side U to V of the graph and by construction all such edges are not part of matching.

- Every even step in the tree will be a matched edge as we are moving from side V to U in the graph and these are the revered edges that were part of matching.

- The highest (deepest) level leaf nodes will have an unmatched vertex from set V. (Ensured by step 2)

3.1.3 DFS on BFS tree

DFS on the generated BFS tree is used to find the disjoint augmenting paths.

1. The source nodes will be highest level leaf nodes of BFS tree.
2. While doing DFS we are moving alternatively between matching and unmatching edges (hence augmenting paths). We keep reversing the direction of these edges as we move to the root and mark vertices visited.
3. If root node is reached, we add source and destination nodes of DFS traversal to the matched vertices set.
4. If it is not possible to reach the root node from some source node, we backtrack and reverse the edges again (to restore direction of edges) and mark the vertices unvisited.
5. Once all source nodes are exhuasted, current iteration is complete.

Note:

- Visited nodes are not visited again in DFS. (This ensures all augmenting paths are disjoint)
- Graph gets reconstructed as per scheme.

3.1.4 Time complexity analysis

Lemma 3.1.1. *Let P be shortest augmenting path wrt M and let P' be an augmenting path wrt $M \oplus P$. Then $|P'| \geq |P| + 2|P \cap P'|$.*

Proof. Let $N = (M \oplus P) \oplus P'$. Note that N is a matching with $|N| = 2 + |M|$. By Lemma 3.0.5, has at least two vertex disjoint augmenting paths P_1 and P_2 with respect to M. As P is a shortest augmenting path with respect to M, we get that $|P_1|, |P_2| \geq |P|$. Since $M \oplus N = P \oplus P'$, we get that $|P \oplus P'| \geq |P_1| + |P_2| \geq 2|P|$. As $|P \oplus P'| = |P| + |P'| - 2|P \cap P'|$. Hence $|P'| \geq |P| + 2|P \cap P'|$. □

Lemma 3.1.2. *Let G be a bipartite graph and let M be a matching in it. Suppose $\mathcal{P} = \langle P_1, P_2, \ldots, P_l \rangle$ is a maximal set of vertex disjoint shortest augmenting paths wrt M. Let $M' = M \oplus P_1 \oplus P_2 \oplus \cdots \oplus P_l$ and let P' be an augmenting paths wrt to M'. Then $|P'| > |P_1| = \cdots = |P_l|$.*

Proof. Let P_i be the last path from $\mathcal{P}$ that intersects P'. Then, P_i is an shortest augmenting path wrt $M_{i-1} = M \oplus P_1 \oplus P_2 \oplus \cdots \oplus P_{i-1}$ and P' is an augmenting path wrt $M_{i-1} \oplus P_i$. By Lemma 3.1.1, we get that $|P'| \geq |P_i| + 2|P' \cap P_i| > |P_i|$. □

Theorem 3.1.3. *The number of iterations of the Hopcraft-Karp algorithm on a bipartite graph on n nodes is at most $2\sqrt{n}$.*

Proof. Suppose we perform $\sqrt{n}$ iterations of the algorithm. If the algorithm halts before performing $\sqrt{n}$ iterations then we are done. If the algorithm does not finish after $\sqrt{n}$ iterations, then we need to consider future iterations. Let M' be a maximum matching and M be the current matching after performing $\sqrt{n}$ iterations. Let, $k = |M'| - |M|$.
By Theorem 3.0.5, there exists at least one augmenting path having length less than $\frac{n}{k}$. Since by Lemma 3.1.2 the augmenting path length monotonically increases in successive iterations, we have

$$\sqrt{n} \leq \frac{n}{k},$$

which implies that $k \leq \sqrt{n}$. Therefore the total number of iterations can be at most $2\sqrt{n}$. □

In each iteration we are performing a BFS and DFS, so by Theorem 3.1.3, the time complexity of Hopcraft-Karp algorithm on a bipartite graph on n nodes and m edges is $O(\sqrt{n}m)$.

3.2 Counting Perfect matchings

Let $G = (V(G), E(G))$ be a graph on n vertices. A matching which covers or spans the whole vertex set of G is called a perfect matching. Notice that in order to have a perfect matching it is necessary that n is even, but it is not sufficient.
Consider the problem of counting the number of perfect matchings in a graph. Will it be easier than counting the number of spanning trees? or harder?. Someone might be tempted to say that finding the number of perfect matching is easier as at first it seems like perfect matchings are less complex structure compare to the spanning trees. Next, in any graph the number of perfect matchings is at most the number of spanning trees (why?). Does merely being less in number implies that find number of perfect matchings is easier? We shortly see that the answer here is No! This some sort of puzzling phenomenon is equivalent to finding the determinant vs find the permanent. The determinant can be solved in $O(n^{2.37})$, whereas, the permanent is

#P-complete problem even though for calculating it checking the signs of product terms is not required.

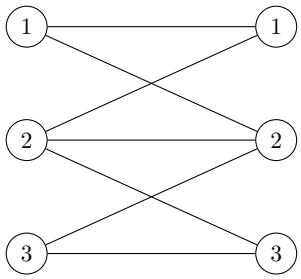


Figure 3.6

As the number of spanning trees is given by any cofactor (a determinant) of the Laplacian matrix it can be computed in $O(n^{2.37})$. However like the permanent of a matrix there is no polynomial time algorithm for finding the number of perfect matchings in a graph. In fact, if there is a polynomial time algorithm (very unlikely), then P=NP. We will establish this fact with the help of bipartite graphs.

3.2.1 Number of perfect matchings in bipartite graph and the permanent

We now consider the number of perfect matchings in a bipartite graph. For a bipartite graph to have a perfect matching both of its parts should have the same number of vertices, such a graph is called balanced bipartite graph. For convenience we will label both the parts as $\{1, \ldots, n\}$ in a balanced bipartite graph on $2n$ vertices. An edge (i, j) means its an edge where vertex i is in the left part, vertex j is in the right part. Check that the perfect matchings in the bipartite graph in Figure 3.6 are

$$\{(1,1),(2,2),(3,3)\},$$
$$\{(1,1),(2,3),(3,2)\},$$
$$\{(1,2),(2,1),(3,3)\}.$$

How to find the number perfect matching in a balanced bipartite graph $G = (X \cup Y, E(G))$? Is the problem 'easy' or 'hard'? Before we answer this let us take a pause here and go back to the square $(0,1)$-matrices. We see that for any such a matrix there is a balanced bipartite graph. Let M be a square $(0,1)$-matrix. We can construct a balanced bipartite graph $G(M)$ with vertex sets $X = \{1, \ldots, n\}, Y = \{1, \ldots, n\}$, and an edge (i, j) is there if and only if $M(i, j) = 1$. For example con-

sider the following matrix

$$M = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix},$$

the $G(M)$ is the bipartite balanced graph shown in Figure 3.6. Conversely, for any balanced bipartite graph $G = (X \cup Y, E(G))$ on $2n$ vertices there is a square matrix whose (i, j)-th entry is 1 if edge (i, j) is in G, else it is 0. This matrix is called the biadjacency matrix of G.
Now we prove a very important result that says that finding the number of perfect matchings is equivalent to calculating the permanent of square $(0, 1)$-matrix.

Theorem 3.2.1. *The permanent of a square $(0, 1)$-matrix M is equal to the number of perfect matchings in $G(M)$.*

Proof. The permanent of M equals the number of nonzero diagonals in M. Each diagonal has n 1s where no two 1s share a row and column. By definition of $G(M)$ a 1 in (i, j)-th position of M correspond to an edge (i, j). As no two 1s in a diagonal share a row and column, each of the corresponding edges does not share a vertex in left or right part. As there n edges these will constitute of perfect matching. Thus diagonals are in one-one correspond to the perfect matchings. Which proves the result. □

Thus we can find the permanent of a square $(0, 1)$-matrix if we can find the number of number of perfect matchings in graph $G(M)$ and vice-versa. We know that the problem of finding the permanent of a square $(0, 1)$-matrix is #P-complete, hence finding the number of perfect matchings in a balanced bipartite graph is #P-complete. As for a general graph finding the number of perfect matching is at least as hard as finding the number of perfect matchings in a balanced bipartite graph. Thus the number of perfect matchings in general graphs is also #P-complete.

Exercise 3.2.2. *Prove that every Eulerian graph has even number of perfect matchings.*

Exercise 3.2.3. *Prove that a graph G has an even number of perfect matchings if and only if there is a nonempty set $S \subseteq V(G)$, such that every vertex in $V(G)$ is adjacent to an even number of vertices in S.*

Chapter 4

Determinant vs Permanent

Ranveer, CSE, IIT Indore

Let

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}.$$

The determinant of A is

$$aei - afh - bdi + bgf + cdh - ceg. \tag{4.1}$$

The permanent of A is

$$aei + afh + bdi + bgf + cdh + ceg. \tag{4.2}$$

Which one is easier to compute?
The determinant and the permanent of an $n \times n$ matrix A, denoted by $\det(A)$, $\mathrm{per}(A)$, respectively, are defined as

$$\det(A) = \sum_{\sigma} sgn(\sigma) a_{1\sigma(1)} \dots a_{n\sigma(n)},$$

$$\mathrm{per}(A) = \sum_{\sigma} a_{1\sigma(1)} \dots a_{n\sigma(n)},$$

where the summation is over all the permutations $\sigma(1), \dots, \sigma(n)$ of $1, \dots, n$, and sgn is 1 or -1 accordingly as σ is even or odd.

1. The determinant of a square matrix may be calculated using its LUP Decomposition.

2. The complexity of the determinant computation of a matrix of order n is the same as the complexity of matrix product of two matrices of order n, that is, $O(n^{\epsilon})$, $2 < \epsilon < 3$. Thus the determinant is polynomial time solvable. (At present $\epsilon = 2.3728$)

3. Permanent computation is #P-complete.

4. The determinant vs permanent problem is equivalent to finding the number of spanning trees vs number of perfect matchings in a bipartite graph.

Chapter 5

Network flow

A flow network $G = (V, E)$ is a directed graph with two special nodes, source s and sink t, and each edge $(u, v) \in E$ has a capacity $c(u, v) \geq 0$.
A flow in G is a real valued function $f : V \times V \to \mathbb{R}$ such that

1. For every edge (u, v), we have, $f(u, v) \leq c(u, v)$. It is called capacity rule. It says that the flow on an edge cannot exceed its capacity.

2. Flow conservation: For every vertex $v \notin \{s, t\}$

$$\sum_{u \in V} f(u, v) = \sum_{w \in V} f(v, w),$$

that is, the amount of flow into a node equals the amount of flow out of it.

Following is a flow network.

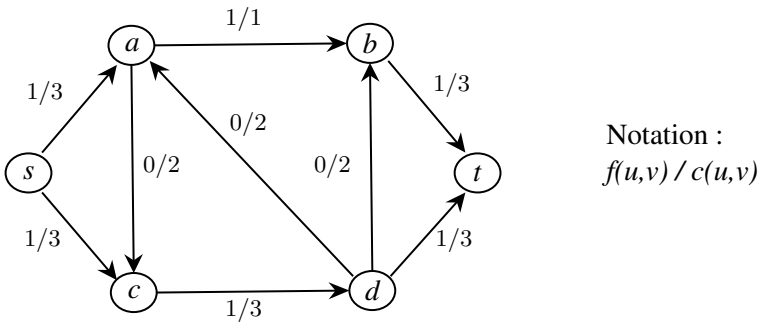


Figure 5.1: A flow network. The flow value is 2.

The value of a flow or flow value in a flow network $G = (V, E)$ is

$$|f| = \sum_{u \in V} f(s, u) = \sum_{u \in V} (u, t).$$

Check that in the flow network in Figure 5.1 the flow value is 2.

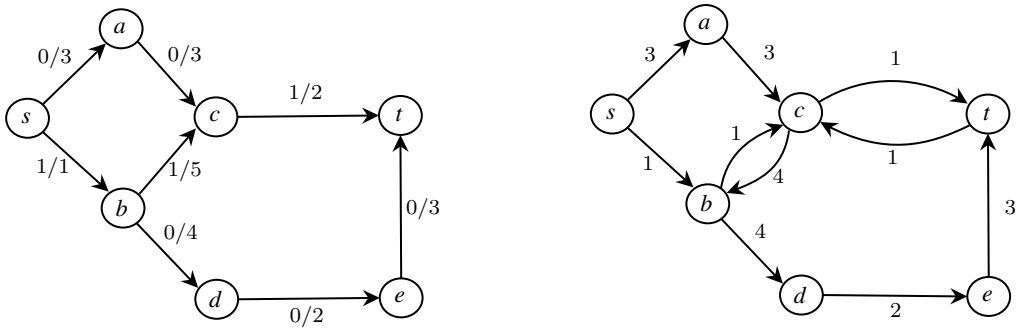
Question 1: Given a flow network G what is a maximum flow from s to t?

5.0.1 Residual Network

Consider an arbitrary flow f in a network G. The residual network G_f has the same vertices as the original network G, and we put one or two edges for each edge (u, v) in G according to the following rule.

1. If $f(u, v) < c(u, v)$, then there is a forward edge (u, v) with capacity $c_f(u, v) = c(u, v) - f(u, v)$.
2. If $f(u, v) > 0$, then there is a backward edge (v, u) with capacity $c_f(v, u) = f(u, v)$.

Example

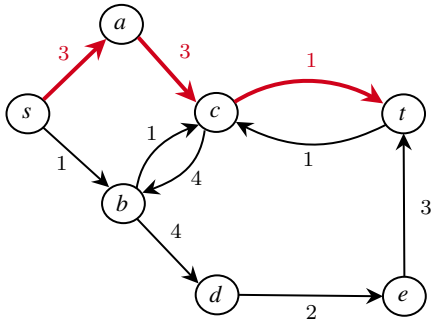


(a) (b)

Figure 5.2: (a) A flow network, (b) the corresponding residual network

5.0.2 Augmenting Path

An augmenting path is a directed path from source s to sink t in the residual network. As the name suggests its purpose is to increase the flow. In the following residual graph a path $P = (s, a, c, t)$ is an augmenting path (shown in red).



The path capacity of a path is the minimum capacity of an edge along that path. For example in the above residual graph the path capacity of the path $P = (s, a, c, t)$ is $\delta(P) = \min\{c(s, a), c(a, c), c(c, t)\} = 1$. Using the augmenting path $P = (s, a, c, t)$, we can increase the flow by $\delta(P) = 1$ unit.

5.0.3 Augmented Path Algorithm (Ford-Fulkerson algorithm, 1950)

Algorithm 3 Ford-Fulkerson Algorithm

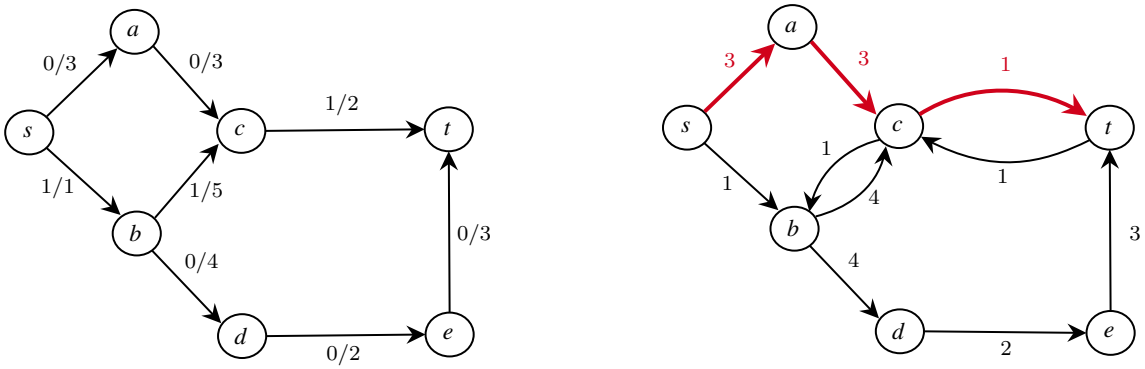
Input: A flow network G
Output: A maximum flow f
Initialize: $f = 0$
while (the residue network G_f contains an augmenting path P in G_f)
{

$$\delta = \min\{c_f(u, v) \mid (u, v) \in P\}$$
augment δ units of flow along P in G and update G_f

}
Return the flow from the final residual network

An illustration:

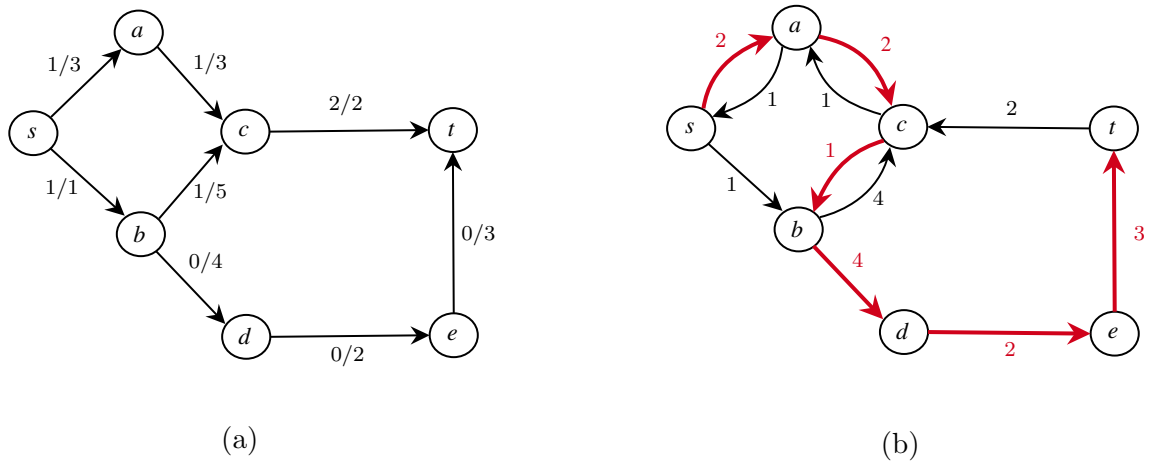
Consider the flow network in Figure 5.3 (a). Currently it has flow value 1. (The algorithm started with flow value 0, we are showing the illustration from an intermediate step.) The corresponding residue network is shown in Figure 5.3 (b), where an augmenting path P is shown in red.



(a) (b)

Figure 5.3: (a) Flow network (b) The residual network. (An augmenting path is shown in red.)

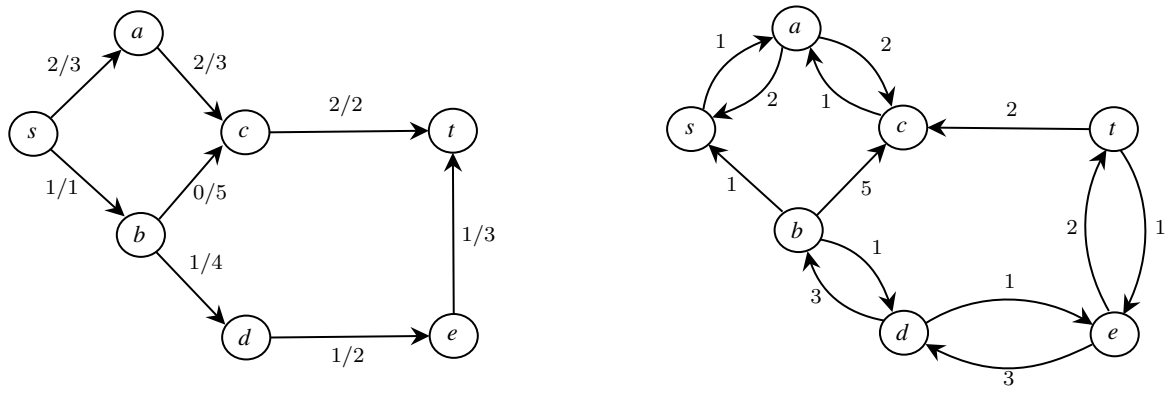
Here, $P = (s, a, c, t)$, $\delta(P) = 1$. Next augmentation is as follows.



(a) (b)

Figure 5.4: (a) Flow network (b) The residual network.

Here $P = (s, a, c, b, d, e, t)$, $\delta(P) = 1$. Next augmentation is as follows.



(a) (b)

Figure 5.5: (a) Flow network (b) The residual network.

Now there are no augmenting paths in the residual network. The algorithm terminates with the flow value 3. It is a maximum flow.

5.0.4 Time complexity

In every iteration we are finding an augmenting path which we can find using DFS or BFS, taking $O(|V| + |E|)$ time. An augmenting path increases the flow at least by 1 units. Thus in worst case the time taken by Ford-Fulkerson algorithm is $O(f^\star(|V| + |E|))$, where, $f^\star$ is the maximum flow value in G.

5.1 Proof of correctness of Ford-Fulkerson Algorithm

To prove the correctness of Ford-Fulkerson Algorithm we need concepts of cut and cut capacity.

A cut $(S, \overline{S})$ is a partition the vertex set V into its subsets S and $\overline{S}$. It gives a set of edges with one endpoint in S and the other in $\overline{S}$. Consider the flow network in Figure 5.6. Let $S = \{s, a, c\}$, $\overline{S} = \{b, d, e, t\}$. This cut consists of the edges $(s, b), (b, c), (c, t)$. A cut $(S, \overline{S})$ is an (s, t) cut if $s \in S$, and $t \in \overline{S}$. For $S = \{s, a, c\}$, $\overline{S} = \{b, d, e, t\}$ the cut is an (s, t) cut.

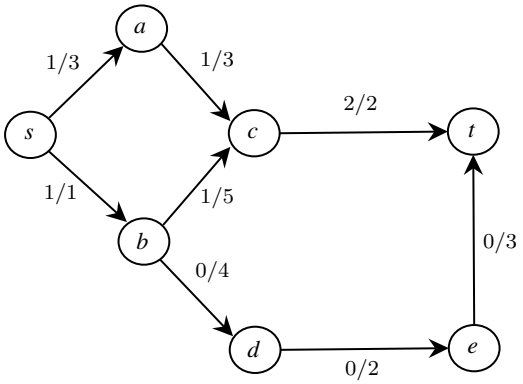


Figure 5.6

The cut capacity of an $(s - t)$ cut $(S, \overline{S})$ is given by

$$c(S, \overline{S}) = \sum_{u \in S, v \in \overline{S}} c(u, v).$$

In the above flow network for $S = \{s, a, c\}$ the cut capacity is

$$c(S, \overline{S}) = c(s, b) + c(c, t) = 1 + 2 = 3.$$

A min-cut is an $(s - t)$ cut having the minimum capacity.

Question 2: Given a flow network G find a min-cut and its capacity?

Net flow across an $(s - t)$ cut

Let $f(e)$ denote the flow on edge e. The net flow across an $(s - t)$ cut $(S, \overline{S})$ is defined to be

$$f(S, \overline{S}) = \sum_{e \text{ out of } S} f(e) - \sum_{e \text{ in to } S} f(e).$$

For example, in the flow network in Figure 5.6 for $S = \{s, a, c\}$

$$f(S, \overline{S}) = f(s, b) + f(c, t) - f(b, c) = 1 + 2 - 1 = 2.$$

Lemma 5.1.1. *Let f be a flow in a flow network G with source s and sink t and let $(S, \overline{S})$ be an $(s - t)$ cut of G, then the flow across $(S, \overline{S})$ is*

$$|f| = f(S, \overline{S})$$

Proof.

$$\begin{aligned}
|f| &= \sum_{e \text{ out of } s} f(e) \\
&= \sum_{v \in S} \Big(\sum_{e \text{ out of } v} f(e) - \sum_{e \text{ into } v} f(e) \Big) \\
&= \sum_{e \text{ out of } S} f(e) - \sum_{e \text{ into } S} f(e) \\
&= f(S, \overline{S})
\end{aligned}$$

□

Lemma 5.1.2. *Let f be any flow and let $(S, \overline{S})$ be a $(s - t)$ cut. Then $|f| \leq c(S, \overline{S})$.*

Proof.

$$\begin{aligned}
|f| &= \sum_{e \text{ out of } S} f(e) - \sum_{e \text{ out into } S} f(e) \\
&\leq \sum_{e \text{ out of } S} f(e) \\
&\leq \sum_{e \text{ out of } S} c(e) = c(S, \overline{S}).
\end{aligned}$$

□

5.2 Max-flow min-cut theorem

Theorem 5.2.1. *If f is a flow in a flow network G with source s and sink t, then the following are equivalent*

1. *f is a max flow in G.*
2. *The residual network G_f contains no augmenting path.*

3. $|f| = c(S, \overline{S})$ *for some* $(s-t)$ *cut* $(S, \overline{S})$.

Proof. (1) $\implies$ (2): Suppose for the sake of contradiction assume that f is max-flow in G but G_f has an augmenting path P. But we can then augment $\delta(P) > 0$ units of flow along P and get a better flow, which contradicts that f is a max-flow.
(2) $\implies$ (3): We can actually construct such an $(s-t)$ cut using the fact that G_f has no augmenting path. Define

$$S = \{v \in V \mid \text{there exists a path from } s \text{ to } v \text{ in } G_f\}.$$

$(S, \overline{S})$ is an $s-t$ cut as $s \in S$ and $t \in \overline{S}$. Now check that

1. For each edge $(u, v) \in (S, \overline{S})$ in G, we have $f(u, v) = c(u, v)$, otherwise, $(u, v) \in E(G_f)$, which would place v in S.

2. For each edge $(v, u) \in (\overline{S}, S)$ in G, we must have $f(v, u) = 0$, otherwise, we would have $(u, v) \in E(G_f)$ with capacity $c_f(u, v) = f(v, u) > 0$, which would place v in S.

Thus we have

$$\begin{aligned} |f| &= \sum_{e \text{ out of } S} f(e) - \sum_{e \text{ out into } S} f(e) \\ &= \sum_{e \text{ out of } S} c(e) - 0 \\ &= c(S, \overline{S}). \end{aligned}$$

(3) $\implies$ (1): By Lemma 5.1.2 $|f| \leq c(S, \overline{S})$ for any $(s-t)$ cut $(S, \overline{S})$. The condition $|f| = c(S, \overline{S})$ thus implies f is max-flow.

□

5.3 Finding a maximum bipartite matching using network flow

We can find a maximum matching in a given bipartite graph by convert it to a suitable flow network and then using flow network algorithms. Let us understand it using an example.

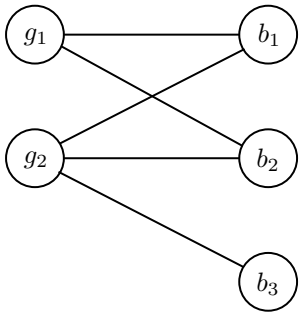


Figure 5.7

Firstly augment the given bipartite graph $G = (L \cup R, E)$ with two vertices source s and sink t, and connect source to set L and set R to sink, and make the directed edges from $s- > L- > R- > t$. Each edge is given the capacity 1.

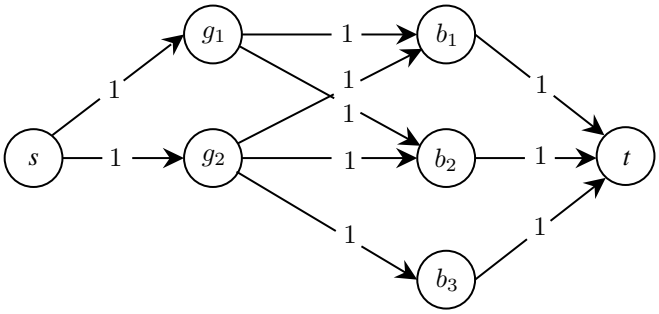
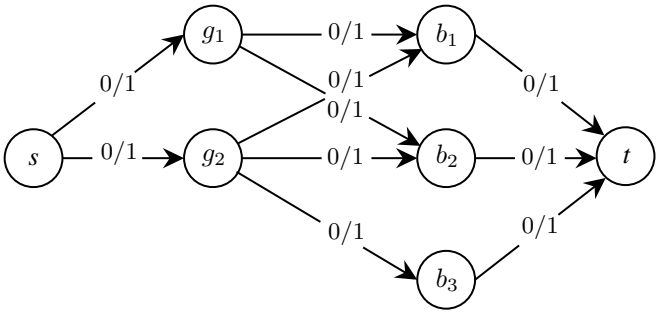


Figure 5.8

Now we will apply the Ford Fulkerson algorithm to find a maximum flow, which will give us a maximum matching. (The proof of correctness is left as an exercise.)

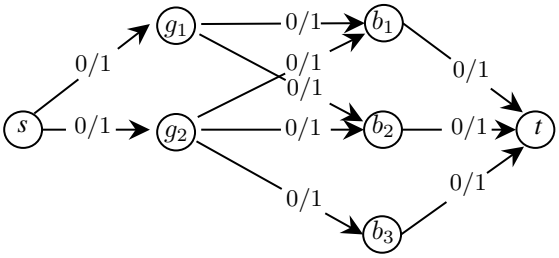
5.3.1 Initialisation



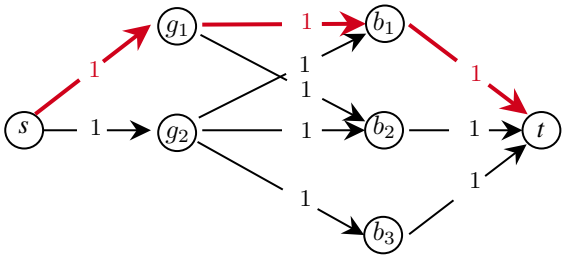
Initialise the network with flow = 0

5.3.2 Working

Iteration 1

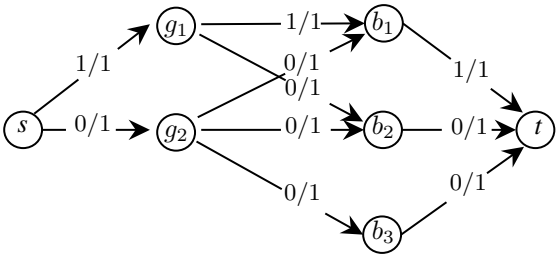


Flow network 1

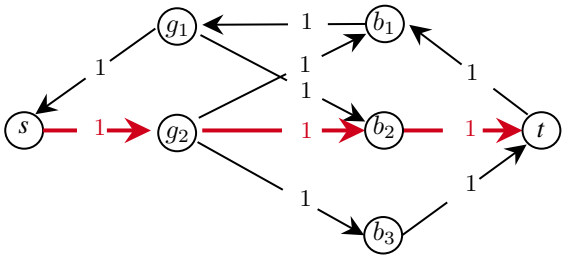


Residual network

Iteration 2

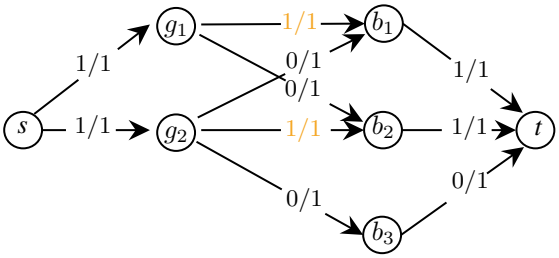


Flow network 2

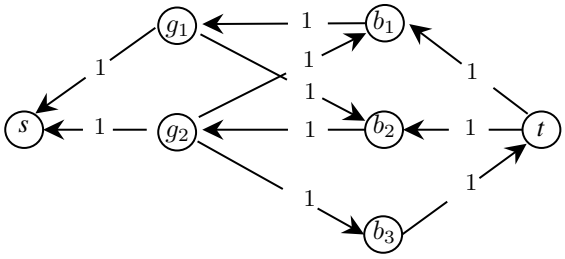


Residual network

Iteration 3



Flow network 3



Residual network

Now STOP! as there is no augmenting path in the residual matrix. This is a maximum matching corresponds to a maximum flow with flow value 2 and the corresponding matching is $M = \{(g_1, b_2), (g_2, b_2)\}$.

Chapter 6

Network flow (The push-relabel algorithm)

Consider a flow network $G = (V, E)$ on n nodes (recall Chapter 5). The augmenting path algorithm always maintains the conservation of fluid at intermediate nodes. However the algorithm that we discuss in this chapter may violate this condition at intermediate nodes. Instead of flow conservation at intermediate nodes, we may have a preflow. A preflow is what we get when we don't have flow conservation at even the non s, t nodes. In a preflow, the flow entering the node is allowed to be more than the leaving it. It is not allowed to be less. So if f is a preflow, then for all

$$v \in V \setminus \{s, t\}, \sum_{(u,v)\in E} f(u, v) \geq \sum_{(v,w)\in E} f(v, w)$$

Example:

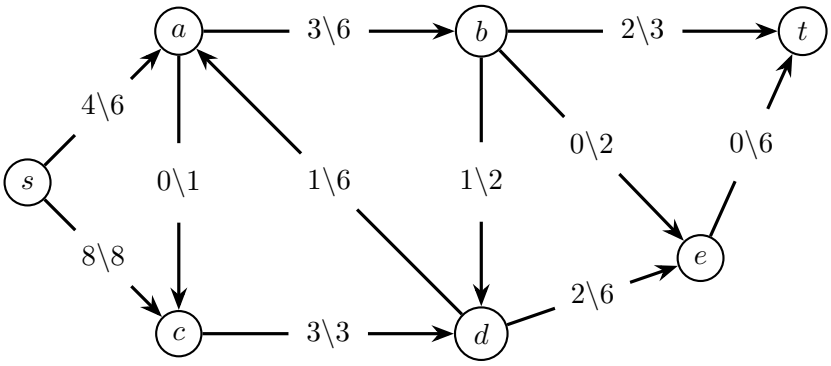


Figure 6.1: A preflow. Nodes a, c, d, e are all active, with excess flow of 2, 5, 1 and 2, respectively

6.0.1 Active node

Given a preflow f, a node v has some access value $e_f(v) \geq 0$, which is just the difference between the flow entering that node and the flow leaving it.

$$e_f(v) = \sum_{(u,v)\in E} f(u,v) - \sum_{(v,w)\in E} f(v,w) \tag{6.1}$$

A node v is active node if $v \notin \{s,t\}$ and $e_f(v) > 0$. In the Figure 6.1 the nodes a, c, d, e are all active, with excesses of 2, 5, 1, and 2, respectively.
Let E_f denote the edges of the residual graph G_f with respect to the preflow f. Recall that, when you send flow f along an (u,v) with capacity c, in the residual graph G_f you get an edge (u,v) with capacity $c-f$, and you get the reverse edge (v,u) with capacity f. If $c = f$, then we drop the edge (u,v) from E_f (since the residual capacity $c - f = 0$). Following is the residual graph corresponding to the preflow in the network shown in Figure 6.1.

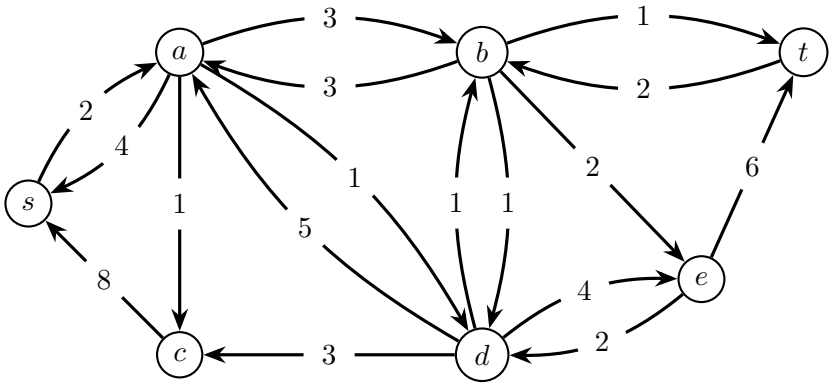


Figure 6.2: Residue network of the flow network in Figure 6.1

6.1 Height (Labeling)

Each node v is given a height (label) $h(v) \geq 0$. The labeling is valid for preflow f if for all edges (u,v) in E_f, we have

$$h(u) \leq h(v) + 1, \tag{6.2}$$

We will maintain the invariant that our current labeling is valid for the current preflow. Call an arc (u,v) admissible if $(u,v) \in E_f$ and

$$h(u) \geq h(v) + 1, \tag{6.3}$$

Since we always have a valid labeling, putting the above two equations together, admissible edges (u, v) in fact satisfy the equality.

$$h(u) = h(v) + 1. \tag{6.4}$$

6.2 The Push-Relabel Algorithm

6.2.1 Initialisation

1. We send $c(s, v)$ flow on each edge $(s, v) \in E$. This gives a valid preflow f_o. (also called the saturation of edges from the source.) At this point, if v is adjacent to s, then $e_{f_o}(v) = c(s, v)$.

2. $h(s) = |V|, h(v) = 0$, for all $v \in V$, $v \neq s$. Note that this is a valid labeling with respect to f_o. In other words, every edge (u, v) in the residual graph G_{f_o} satisfies $d(u) \leq d(v) + 1$.

6.2.2 The main loop

While there is an active node u, do one of the following for u

- **Push(u):** If $\exists v$ with admissible arc $(u, v) \in E_f$, then send flow $\delta = \min(e_f(u), c_f(u, v))$ from u to v. If $\delta = c_f(u, v)$ this is called a saturating push, else it is an non-saturating push.

- **Relabel(u):** If for all edges (u, v), $h(v) \geq h(u)$, then set $h(u) = 1 + \min_{(u,v) \in E_f}(h(v))$

Illustration:

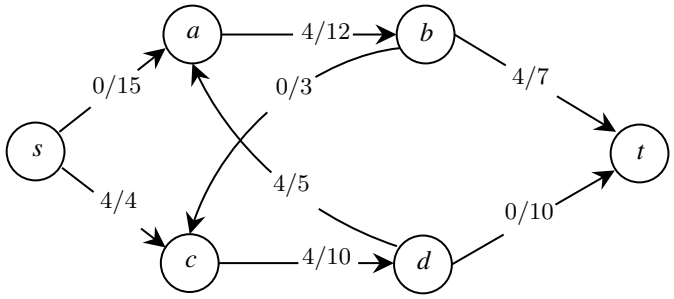
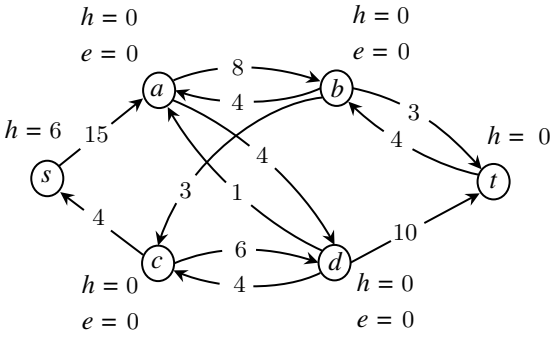
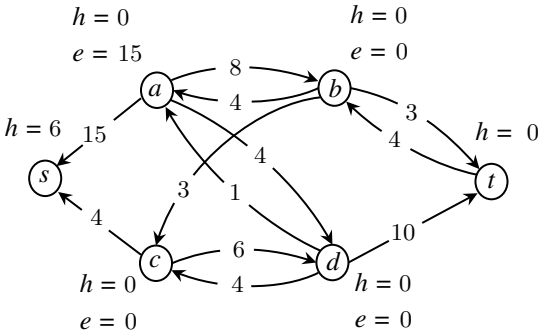


Figure 6.3: A flow network G with a flow f.

In the following figures (a) is the residue network after the initialising the labels (b) is the residue network after saturating edges out of source s.

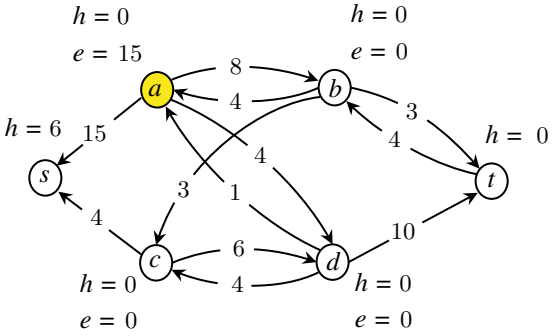


(a)

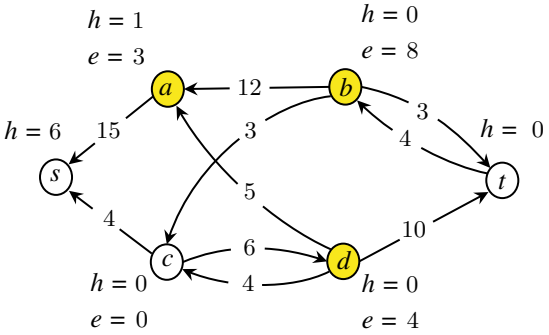


(b)

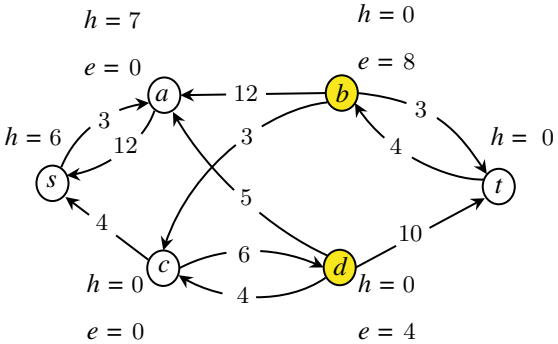
Following figures show the residue networks obeying the main loop of the push-relabel algorithm.



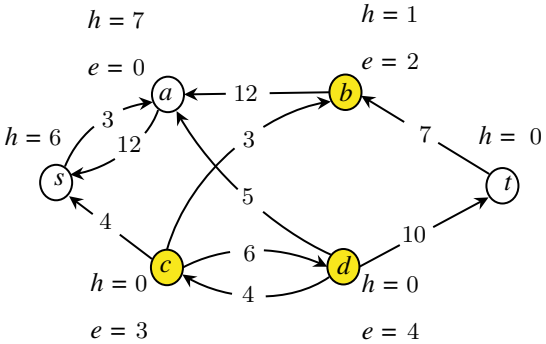
(a)



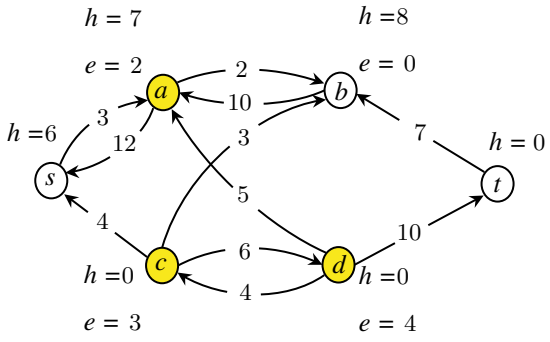
(b)



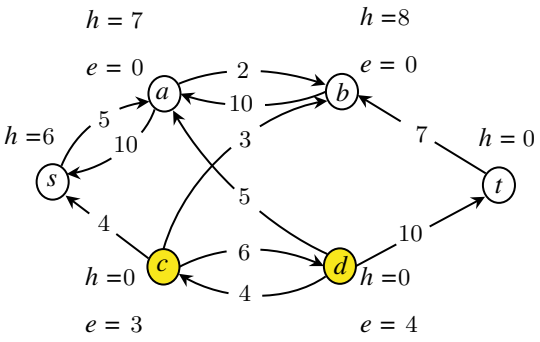
(a)



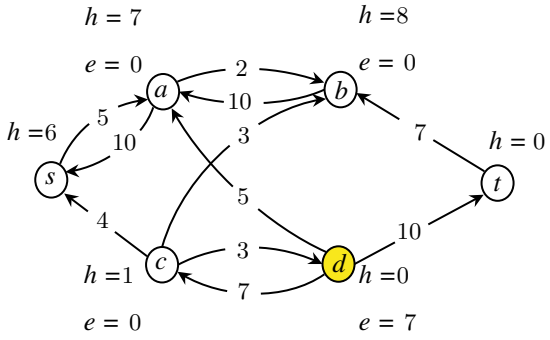
(b)



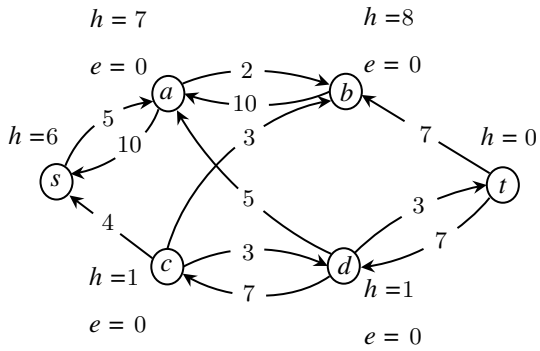
(e)



(f)



(g)



(h)

At (h) we stop as there is no active node.

6.2.3 The proof of correctness

Lemma 6.2.1. *The label h remain valid with respect to the current preflow f.*

Proof. The proof is by induction. Initially h is valid with respect to f_0 (saturating edges out of s). When we send flow along an edge (u, v), this adds the residual edge (v, u). But since we just sent flow on the admissible edge (u, v), we know that $h(u) = h(v)+1$, so the new edge $(v, u) \in E_f$ satisfies $h(v) = h(u)-1 \leq h(u)+1$, and the labels remain valid. (Note that during relabeling for all edges (u, v) out of $u, h(u) \leq h(v)$, then we set $h(u)=1 + \min_{v:(u,v)\in E_f} h(v)$.) □

Theorem 6.2.2. *Suppose the algorithm terminates, then the final preflow is a maximum flow.*

Proof. If the algorithm terminates with flow f, then none of the nodes in $V \setminus \{s, t\}$ is active, and these nodes have no excess fluid. There is flow-conservation at all the intermediate nodes. Hence f is a flow.

Now suppose this is not a maximum flow. Then we know that there must be an augmenting path, that is, a path from s to t in the residual graph G_f. Let this path be $v_0 = s, v_1, v_2, v_3, \ldots, v_k = t$. Since the final labeling h must be valid, we know that each $h(v_i) \leq h(v_{i+1}) + 1$. Hence, $h(s) = h(v_0) \leq h(t) + k$. But the path can have at most $n - 1$ edges, so $k \leq (n - 1)$, and hence $h(s) \leq h(t) + n - 1$. But this contradicts the fact that $h(s) = n$, and $h(t) = 0$. □

Exercise 6.2.3. *Prove that at each step of the push-relabel algorithm, every vertex has height at most* $2|V| - 1$.

Exercise 6.2.4. *Prove that the push-relabel algorithm executes at most* $(|V|-2)(2|V|-1) \leq 2|V|^2$ *relabel operations.*

For more details see here and here

Chapter 7

Linear programming and duality

A linear-programming is the problem of either minimizing or maximizing a linear function subject to a finite set of linear constraints. A linear program in ***standard*** form is the maximization of a linear function subject to linear inequalities.

7.1 Standard Form

A linear program in its standard form looks as follows. There is an linear objective function

$$c_1x_1 + \ldots + c_nx_n$$

whose value we wish to maximize, where, $c_1, \ldots, c_n$ are real constants and $x_1, \ldots, x_n$ are nonnegative variables. Furthermore, there are following m linear constraints on $x_1, \ldots, x_n$

$$\sum_{j=1}^{n} a_{ij}x_j \leq b_i \quad \text{for } i = 1, 2, \ldots, m,$$

where, b_i, and a_{ij} are real constants for $i = 1, \ldots, m$ and $j = 1, \ldots, n$. We can write the linear program in matrix form as follows.
maximize

$$c^T x \tag{7.1}$$

subject to

$$Ax \leq b \tag{7.2}$$

$$x \geq \mathbf{0}, \tag{7.3}$$

where, $c = (c_1, \ldots, c_n)^T, x = (x_1, \ldots, x_n)^T, b = (b_1, \ldots, b_m)^T, A = (a_{ij})_{m \times n}$.
A linear program might not be in standard form due to any of the following reasons

- Minimization rather than maximization.
- Variables might have negativity constraints.
- There may be equality constraints.
- Greater than or equal to inequality constraints.

We can convert any linear program in standard.
Consider a linear program L

$$\begin{aligned} \text{minimize } & -2x_1 + 3x_2 \\ \text{subject to } & x_1 + x_2 = 7 \\ & x_1 - 2x_2 \leq 4 \\ & x_1 \geq 0 \end{aligned}$$

We can convert the objective function as a maximization function just by negating the coefficients.

$$\begin{aligned} \text{maximize } & 2x_1 - 3x_2 \\ \text{subject to } & x_1 + x_2 = 7 \\ & x_1 - 2x_2 \leq 4 \\ & x_1 \geq 0 \end{aligned}$$

Check that x_2 does not have a nonnegativity constraint, so the program is still not in a standard form.
Replace each variable x_j that does not have nonnegativity constraint with $x_j - x_j^{''}$ and add the negativity constraints $x_j \geq 0, x_j^{''} \geq 0$.
If $\overline{x}$ was the feasible solution to the original program, any feasible solution $\widehat{x}$ to the new linear program will have

$$\overline{x}_j = \widehat{x}_j^{'} - \widehat{x}_j^{''}$$

with the same objective value. Also, any feasible solution $\overline{x}$ to the original linear program corresponds to a feasible solution $\widehat{x}$ to the new linear program with $\widehat{x}_j = \overline{x}_j$ and $\overline{x}_j^{''} = 0$ if $\overline{x}_j \geq 0$, or with $\widehat{x}_j^{''} = \overline{x}_j$ and $\overline{x}_j = 0$ if $\widehat{x} < 0$. The resulting LP is

$$\begin{aligned} \text{maximize } & 2x_1 - 3x_2^{'} + 3x_2^{''} \\ \text{subject to } & x_1 + x_2^{'} - x_2^{''} = 7 \\ & x_1 - 2x_2^{'} + 2x_2^{''} \leq 4 \\ & x_1, x_2^{'}, x_2^{''} \geq 0, \end{aligned}$$

It is still not in standard form as there is a equality constraint. We can remove the equality constraint by replacing it with one 'less than equal to' and one 'greater than

equal to' constraints

$$\begin{aligned} \text{maximize} \quad & 2x_1 - 3x_2^{'} + 3x_2^{''} \\ \text{subject to} \quad & x_1 + x_2^{'} - x_2^{''} \leq 7 \\ & x_1 + x_2^{'} - x_2^{''} \geq 7 \\ & x_1 - 2x_2 + 2x_2^{''} \leq 4 \\ & x_1, x_2^{'}, x_2^{''} \geq 0 \end{aligned}$$

Finally we can write it in the following equivalent form, which is a standard form.

$$\begin{aligned} \text{maximize} \quad & 2x_1 - 3x_2^{'} + 3x_2^{''} \\ \text{subject to} \quad & x_1 + x_2^{'} - x_2^{''} \leq 7 \\ & x_1 + x_2^{'} - x_2^{''} \leq 7 \\ & x_1 - 2x_2^{'} + 2x_2^{''} \leq 4 \\ & x_1, x_2^{'}, x_2^{''} \geq 0 \end{aligned}$$

Or equivalently we can write

$$\begin{aligned} \text{maximize} \quad & 2x_1 - 3x_2 + 3x_2 \\ \text{subject to} \quad & x_1 + x_2 - x_3 \leq 7 \\ & -x_1 - x_2 + x_3 \leq 7 \\ & x_1 - 2x_2 + 2x_3 \leq 4 \\ & x_1, x_2, x_3 \geq 0. \end{aligned}$$

7.1.1 Slack Form

A linear program in ***slack*** form is the maximization of a linear function subject to linear equalities. It is a form where the non negativity constraints are the only inequality constraints, and remaining constraints are equalities. Let,

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i.$$

We can replace it with two equality constraints,

$$s_i = b_i - \sum_{j=1}^{n} a_{ij} x_j$$

$$s_i \geq 0.$$

We write,

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j,$$

with non negativity constraint $x_{n+i} \geq 0,\ i = 1, \ldots, m$.

For example, LP in 7.1 has the slack form

$$\begin{aligned}
\text{maximize} \quad & 2x_1 - 3x_2 + 3x_3 \\
\text{subject to} \quad & x_4 = 7 - x_1 - x_2 + x_3 \\
& x_5 = -7 + x_1 + x_2 - x_3 \\
& x_6 = 4 - x_1 + 2x_2 - 2x_3 \\
& x_1, x_2, x_3, x_4, x_5, x_6 \geq 0
\end{aligned}$$

Variable on the left hand side of equalities are called basic variables and those on the right hand side are non basic variables. In the slack form we often write (omit) maximize function and introduce a new variable z for it, and also omit non negative constraints.

$$\begin{aligned}
\text{maximize} \quad & z = 2x_1 - 3x_2 + 3x_3 \\
\text{subject to} \quad & x_4 = 7 - x_1 - x_2 + x_3 \\
& x_5 = -7 + x_1 + x_2 - x_3 \\
& x_6 = 4 - x_1 + 2x_2 - 2x_3
\end{aligned}$$

7.1.2 The Simplex Algorithm

The simplex algorithm takes as input a linear program and returns an optimal solution. Consider an example in Slack form:

$$\begin{aligned}
\text{maximize} \quad & z = 3x_1 + x_2 + 2x_3 \\
\text{subject to} \quad & x_4 = 30 - x_1 - x_2 - 3x_3 \\
& x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \\
& x_6 = 36 - 4x_1 - x_2 - 2x_3
\end{aligned}$$

Given 3 equations, 6 variables (infinite solutions). A basic solution is put all non basic variables as 0 and we get basic solution is $(0, 0, 0, 30, 24, 36)$ and the objective value is $z = 0$.

Improvement: Select a non basic variable x_e whose coefficient in the objective function is positive, and increase it without violating any of the constraints. Choose x_1, it can be increased at most 9 (check that the tightest constraint is given by the third equation).

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \tag{7.4}$$

we rewrite the other equations and the objectives. x_1 is now a basic variable (on left)

while x_6 is nonbasic variable (on right). The resulting equivalent form is

$$\text{maximize} \quad z = 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4} \tag{7.5}$$

$$\text{subject to} \quad x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \tag{7.6}$$

$$x_4 = 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \tag{7.7}$$

$$x_5 = 6 - \frac{3x_2}{4} - 4x_3 + \frac{x_6}{4} \tag{7.8}$$

$$\tag{7.9}$$

Set nonbasic variables to 0 to get the basic solution. It gives the objective value 27. Now, select nonbasic variable, such that objective function may increase. Continuing this the final LP will be

$$\text{maximize} \quad z = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \tag{7.10}$$

$$\text{subject to} \quad x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \tag{7.11}$$

$$x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \tag{7.12}$$

$$x_3 = 18 - \frac{x_3}{2} - \frac{x_5}{2} \tag{7.13}$$

Set nonbasic variables to 0 to get the basic solution. It gives the objective value 28. Now since the variables x_3, x_5, x_6 are all nonnagative and their coefficients in the objective function are negative, we cannot increase the objective value. So, 28 will be the optimal value.

7.2 Linear-programming duality

Consider the general linear program given in 7.1–7.3 in a standard form. The aim of this linear program is to maximize the objective function $c^T x$. We now define a related minimization problem such that the minimum value of its objective function is the same as the maximum value of $c^T x$. The minimization problem is called the dual of the original maximization problem. When referring to the dual program, we call the original program the primal.

Given a primal linear program as in 7.1-7.3, we define the dual linear program as minimize

$$b^T y \tag{7.14}$$

subject to

$$A^T y \geq c \tag{7.15}$$

$$y \geq \mathbf{0}, \tag{7.16}$$

Now we prove a important result relating the feasible solutions of primal and its dual.

Theorem 7.2.1. *Let $\tilde{x}$ be any feasible solution of the primal linear program in 7.1–7.3 and let $\tilde{y}$ be any feasible solution of the dual linear program in 7.14–7.16. Then we have*

$$c^T \tilde{x} \leq b^T \tilde{y}.$$

Proof. We have

$$\begin{aligned} c^T \tilde{x} &\leq (A^T \tilde{y})^T \tilde{x} \qquad \text{by 7.15} \\ &\leq \tilde{y}^T A \tilde{x} \\ &\leq \tilde{y}^T b \qquad \text{by 7.2} \\ &= b^T \tilde{y}. \end{aligned}$$

It proves the result. □

This gives the following important corollary.

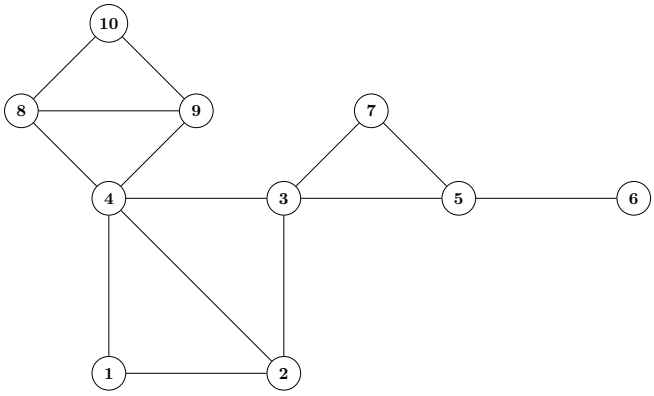
Corollary 7.2.2. *If a feasible solution $\bar{x}$ of a primal LP has the same objective value as the objective value a feasible solution $\bar{y}$ of its dual LP, then this objective value is the optimal value for primal LP as well as its dual LP.*

Chapter 8

Finding Cut-vertices and Biconnected components

Dr. Ranveer

Let $G = (V, E)$ be an undirected graph, with vertex-set V and edge set E. Graph G is called *connected* when there is a path between every pair of vertices. A maximally connected subgraph of G is called a *connected component* or simply *component*. A *cut-vertex* in an undirected graph G is a vertex whose deletion creates more connected components than previously in the graph. In the following graph vertices $3, 4, 5$ are the cut-vertices.



8.1 Finding cut-vertices

Naive Approach: For every vertex $v \in G$: remove v from the corresponding component and check if the component remains connected (use Breadth-first search or Depth-First search). If the resulting subgraph is disconnected, add v to cut-vertex

list.

Complexity: The time complexity of naive approach is $O\big(|V| \times (|V| + |E|)\big)$. Can we do better ?

8.1.1 Key observation

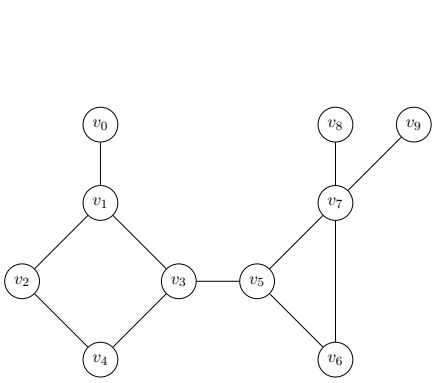
A vertex v in G is a cut-vertex if and only if there exist two other vertices x and y such that every path between x and y passes through v; in this case and only in this case the deletion of v from G destroy all paths between x and y. This observation allow us to use depth-first search to find the cut-vertices of G in $O(|V| + |E|)$ operations. Let us first recall what DFS is.

Depth-first Search (DFS)

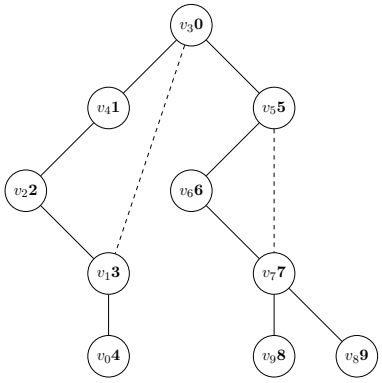
1. Explore deeper in the graph whenever possible
2. Edges are explored out of the most recently discovered vertex v that still has unexplored edges
3. When all the incident edges of v have been explored, backtrack to the vertex from which v was discovered

Depth-first spanning tree

A DFS traversal gives us a tree which we call *depth-first spanning tree*.



G



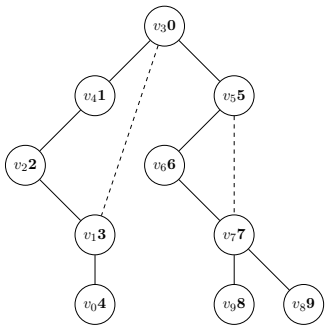
Depth-first spanning tree of G

In a Depth-first spanning tree, *depth first number (dfn)* of a vertex v, denoted by $dfn(v)$ is the discovery time of v (shown on the right side of the vertices in bold). Dashed edges are the back edges. The following table note down the depth first number of eac vertex.

Vertex	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
dfn	4	3	2	0	1	5	6	7	9	8

8.2 Determining cut vertices

1. The root of a DFS-tree is a cut-vertex if and only if it has at least two children. (v_3 is a cut-vertex.)



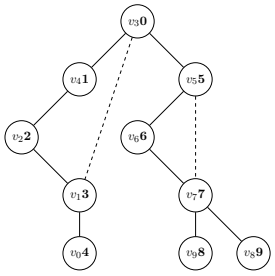
2. A nonroot vertex v of a DFS-tree is a cut-vertex of G if and only if v has a child s such that there is no back edge from s or any descendant of s to a proper ancestor of v.

Notation: Let T be a DFS-tree, $(v, u) \in T$ means v is the parent of u. B is the set of all the back edges.

low function

We define $low(v)$ as the smallest value of $dfn(x)$, where x is a vertex in DFS-Tree T that can be reached from v by following zero or more tree edges followed by at most one back edge. In other words,

$$low(v) = min\Big(\{dfn(v)\} \cup \{low(x)|(v,x) \in T\} \cup \{dfn(x)|(v,x) \in B\}\Big).$$



The following table notes down the low values of vertices.

Vertex	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
dfn	4	3	2	0	1	5	6	7	9	8
low	4	0	0	0	0	5	5	5	9	8

Theorem 8.2.1. *Let $G = (V, E)$ be connected graph with a* DFS-*Tree T and with back edges B. Then $a \in V$ is a cut-vertex if and only if there exist vertices $v, w \in V$ such that v is a child of a in T, w is not a descendant of v in T and $low(v) \geq dfn(a)$.*

Example: $(v_3, v_4), low(v_4) = dfn(v_3)$ so v_3 is a cut-vertex. $(v_6, v_7), low(v_7) < dfn(v_6)$ so v_6 is not a cut-vertex.

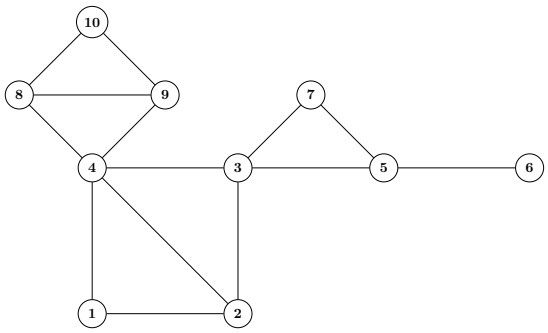
8.2.1 Algorithmic steps for cut-vertices

1. During DFS, calculate the values of the *low* function for every vertex.
2. After we finish the recursive search from a child u of a vertex v, we update $low(v)$. Vertex v is a cut-vertex, disconnecting u, if $low(u) \geq dfn(v)$.
3. When encountering a back-edge (v, u) update $low(v)$ with the value of $dfn(u)$
4. If vertex v is the root of the DFS tree, check whether w is its second child

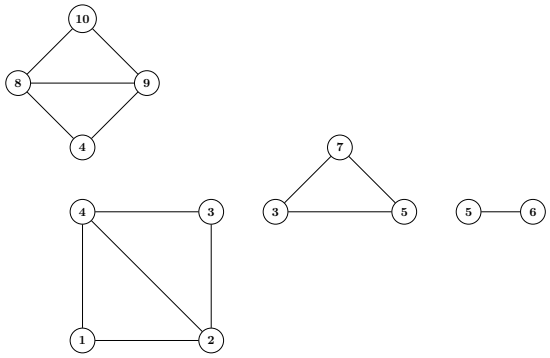
Complexity: Since the algorithm is a depth-first search, the time required is $O(|V| + |E|)$.

8.3 Blocks (or biconnected components or 2-connected components)

A *block* in a graph G is a maximally connected subgraph that has no cut-vertex. For example consider the graph



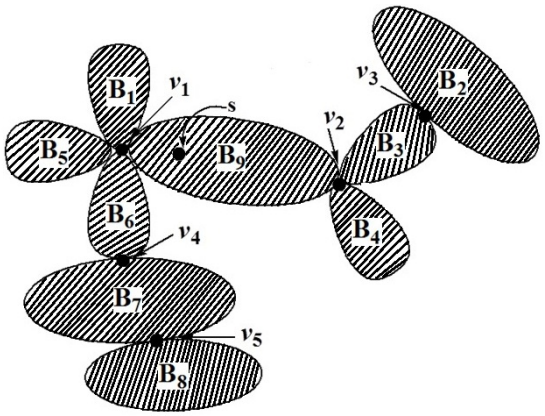
its blocks are the following



How to find blocks in any undirected graph ?

8.3.1 Finding blocks

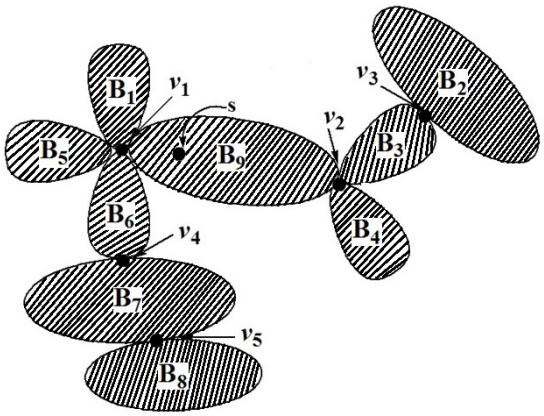
Recognizing cut-vertices, using DFS, we can determine the biconnected components by storing the edges on a stack as they are traversed. HOW?



$v_1, \ldots, v_5$ are the cut-vertices. $B_1, \ldots, B_9$ are the block.

Central Idea

Let us start DFS at vertex s in B_9. Next, suppose we wish to go into B_4 by passing through v_2. By DFS nature all the edges in B_4 must be traversed before we back up to v_2. Now if we leave B_4 and go into B_3 and then into B_2 through v_3. If we store the edges in a stack, by the time we pass through v_3 back into B_3, all the edges of B_2 will be on top of the stack, and forms a biconnected component. When they are removed, the edges on the top of the stack will be from B_3, and we will once again be traversing B_3.



8.4 Algorithmic steps for biconnected components

1. During DFS, use a stack to store visited edges (tree edges or back edges)
2. After we finish the recursive search from a child u of a vertex v, we check if v is a cut-vertex for u. If it is, we output all edges from the stack until (v, u). These edges form a biconnected component
3. When we return to the root of the DFS-tree, we have to output the edges even if the root is not a cut-vertex (graph may be biconnected).

Complexity: Since the algorithm is a depth-first search with a constant amount of extra work done as each edge is traversed, the time required is $O(|V| + |E|)$.

References

1. Hopcroft, J. and Tarjan, R., 1973. Algorithm 447: efficient algorithms for graph manipulation. Communications of the ACM, 16(6), pp.372–378.
2. Reingold, E.M., Nievergelt, J. and Deo, N., 1977. Combinatorial algorithms: theory and practice. Prentice Hall College Div.

Bibliography

[1] R. B. Bapat. *Graphs and Matrices*. Springer, 2014.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2022.

[3] D. M. Cvetkovic, C. DM, et al. Spectra of graphs. theory and application. 1980.

Chapter 9

Fibonacci Heaps (Fredman and Tarjan, 1984)

A **Fibonacci heap** is a collection of rooted trees that are min-heap ordered. That is each tree obeys the min-heap property; the key of a node is greater than or equal to the key of its parent.

Following is a Fibonacci heap.

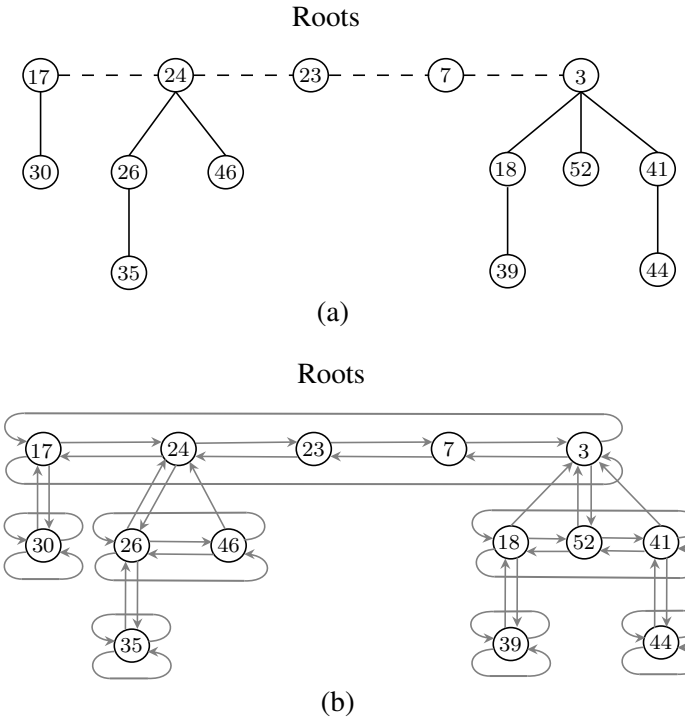


Figure 9.1: (a) A Fibonacci heap. There are five min-heap ordered trees. (b) Pointer topology for the Fibonacci heap in (a).

9.1 Attributes

1. Each node x contains a pointer $x.p$ to its parent and a pointer $x.child$ to any one of its children.

2. Children of x are linked together in a circular doubly linked list. (Also called **child list** of x)

3. Each child y in a child list has pointers $y.left$ and $y.right$ that point to y's left and right siblings, respectively (the order does not matters). If y has no siblings, then $y.left = y.right = y$.

(Why circular doubly linked lists? What are the benefits of circular doubly linked list?)

9.1.1 Some more node attributes:

1. $x.degree$:= The number of children in the child list of x.

2. $x.mark$:= It's a Boolean-valued attribute which indicates whether a node x has lost a child since the last time it was made the child of another node. All the new node are marked false (unmarked), also a node becomes unmarked whenever it is made the child of another node. (importance of marking will be clear when we perform a decrease-key operation.)

3. We access the Fibonacci heap H by a pointer $H.min$ to the root of a tree containing the minimum key (it is the minimum node of H).

   If there are more than one root having minimum key, any of these node serve the purpose. If H is empty, then $H.min$ is NIL.

4. Like child list of a node, the roots of all the trees in a Fibonacci heap H are linked together using their left and right pointers into a circular doubly liked list called the root-list of H and $H.min$ points to the node in root list whose key is minimum (order does not matters).

5. $H.n$:= the member of nodes in H currently.

9.2 Potential function

In a given Fibonacci heap H, we indicate by $t(H)$ the number of trees in the root list of H, and by $m(H)$ the number of marked nodes in H. We define the potential $\phi(H)$

of Fibonacci heap H by

$$\phi(H) = t(H) + 2m(H)$$

The potential of the Fibonacci heap given in Figure 9.1 is $5 + 2 \times 3 = 11$.

9.3 Heap Operations

We will consider insertion, find min, merging two heaps, delete min, extract min operations on Fibonacci heaps.

9.3.1 Inserting a node into a Fibonacci heap:

1. Create a new tree with just one node (a singleton tree).
2. Update min pointer (if necessary) after adding the new singleton tree.

See an example below.

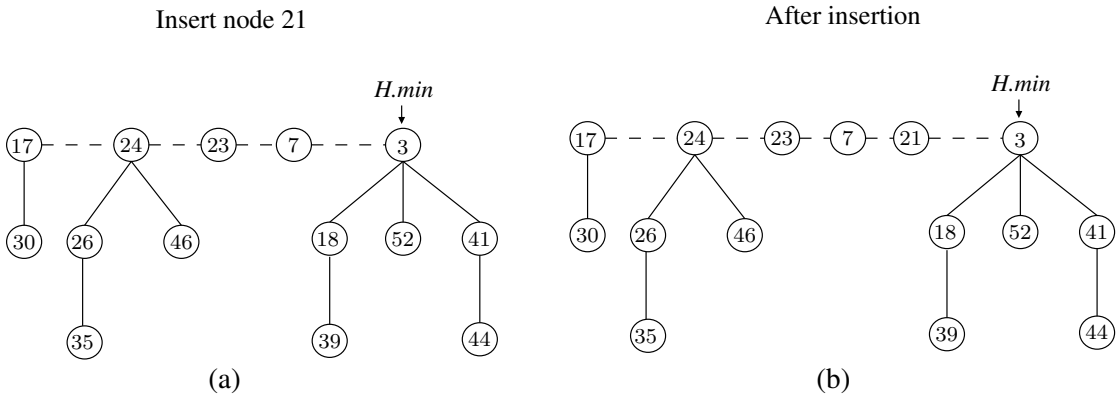


Figure 9.2: A node with a key value 21 is added to (a) resulting in (b). Since 21 is larger than the current minimum value $H.min$ does not change.

Amortized Cost: The amortized cost is actual time plus change in potential. Let H be the initial Fibonacci heap and H' be the resulting Fibonacci heap after inserting a node. Then, $t(H') = t(H) + 1$ and $m(H) = m(H')$, so change in potential is

$$((t(H) + 1 + 2m(H)) - (t(H) + 2m(H)) = 1.$$

Hence the amortized cost is $O(1) + O(1) = O(1)$.

9.3.2 Finding the minimum node in Fibonacci heap

The minimum node is given by $H.min$, so we can find the minimum node in $O(1)$ actual time. Since the potential does not change, the amortized cost of this operation

is equal to its $O(1)$ actual cost.

9.3.3 Uniting two Fibonacci heaps H_1 and H_2

1. Concatenate the root lists of H_1 and H_2
2. Determine the new minimum node. In the process H_1 and H_2 will be destroyed. The change in potential is zero.

9.3.4 Deleting the minimum node from a Fibonacci heap (or Extracting minimum node)

1. Delete the minimum node (node where $H.min$ points), add its children to the root-list.
2. Consolidate the root-list by **linking roots** of equal degree until at most one root remains of each degree. Linking roots of two min-heap trees means making the larger root be a child of smaller root.

Following is an example of linking roots.

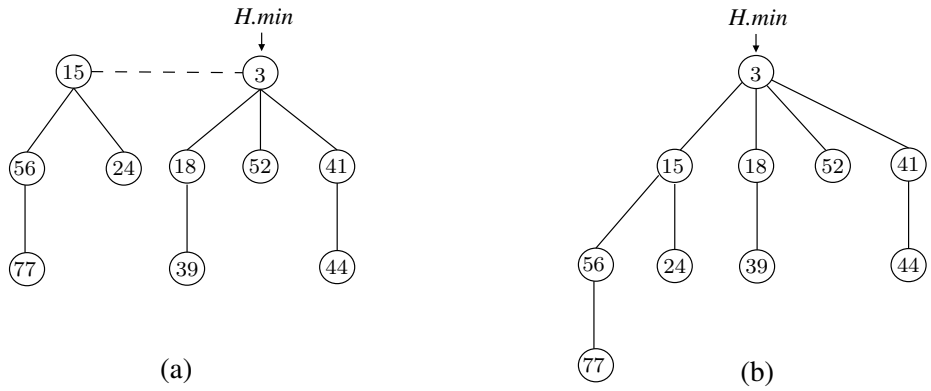


Figure 9.3: The tree in (b) is resulting after linking two roots in (a)

Consolidation:

1. Find two roots x and y in the root list with the same degree. WLOG let $x.key \leq y.key$
2. Link y to x: Remove y from the root list and make y a child of x. This will increase degree of x by 1 and clears the mark on y.

To keep track of degrees of roots in the root list we need an array A. The i-th cell $A(i)$ will have a pointer to a node having degree $A(i)$.

An illustration (see captions for some description).

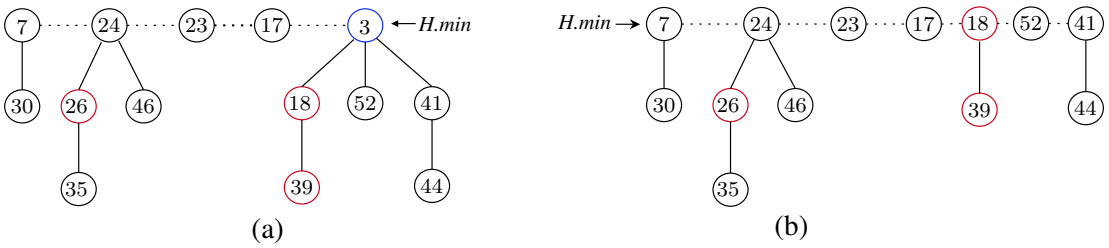
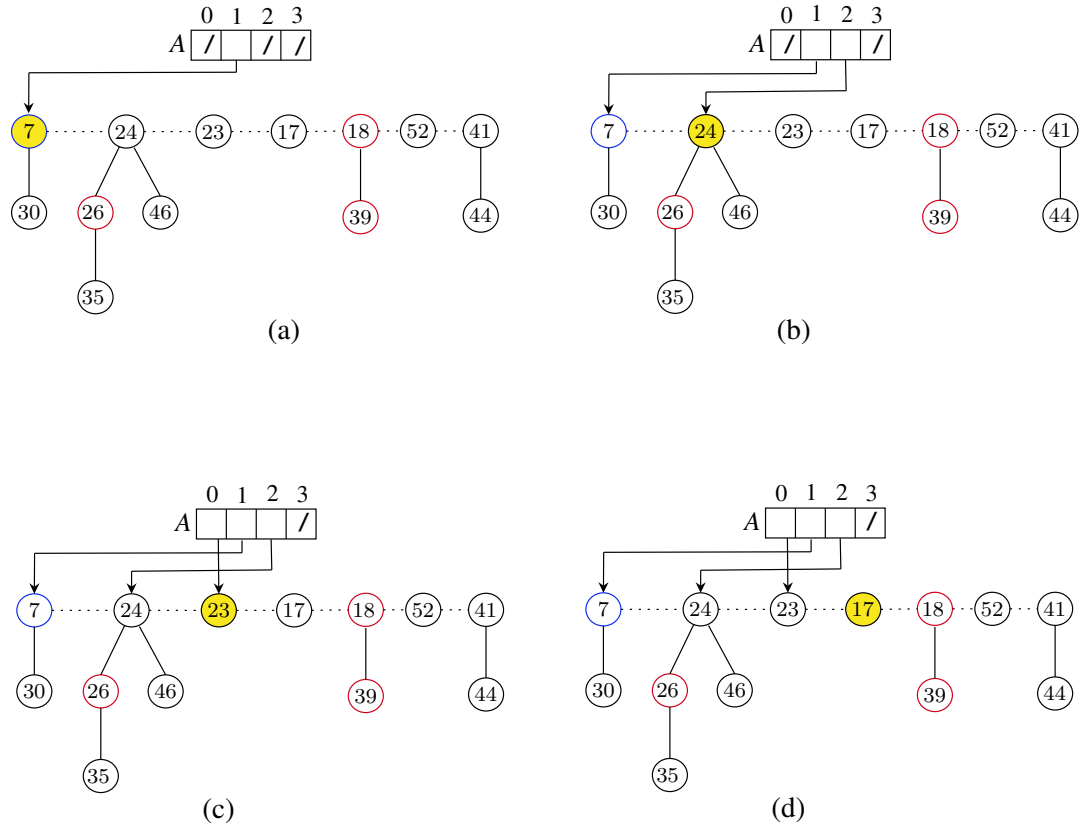
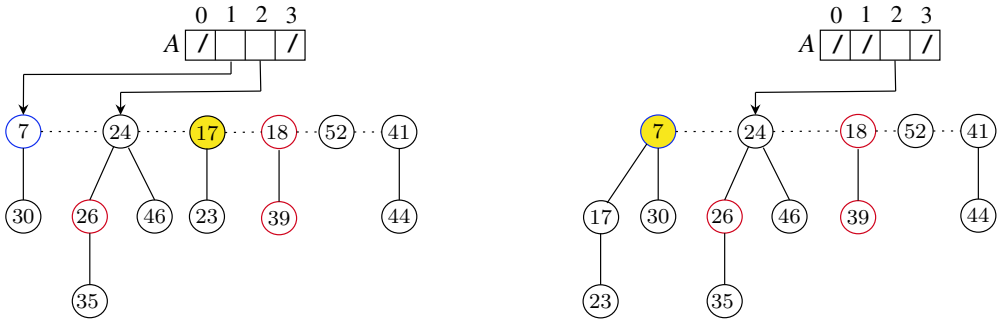


Figure 9.4: Cut the minimum node 3 in (a) move the min pointer to right and put it's children to the root list resulting in (b).

Start maintaining the degree array A. Now onward the current min pointer is shown in yellow shaded node (the current min pointer might not be at the minimum node during the intermediate steps).



Now the current pointer is at node 17 which is having degree 0. But already 23 is mapped to the cell with 0 degree. So we have to link 23 to 17, shown in (e).



Then further 17 will be linked to 7 as shown in (f). Now continuing this way we get the following Fibonacci heap.

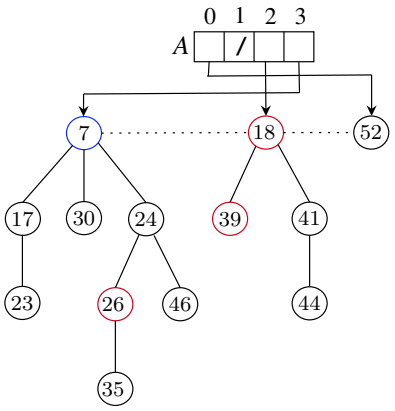


Figure 9.5

Question: Show that the amortized cost of extracting the minimum node of an n-node Fibonacci heap is $O(D(n))$.

9.3.5 Decreasing a key and deleting a node in H

1. If heap-order is not violated, just decrease the key of node x.

2. Otherwise, cut tree (sub-tree) rooted at x and add it to the root list and mark it's parent.

3. As soon as a node has it's second child cut, cut it off and add it to the root list and unmark it (do cascading till root).

Here is an example.

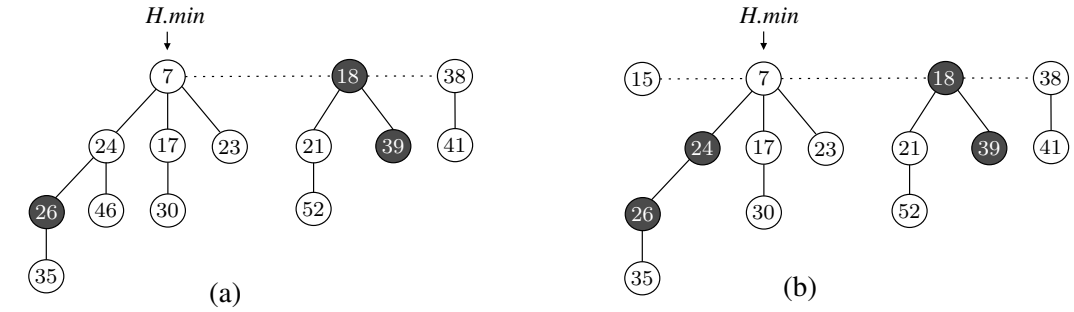


Figure 9.6: (a) Decrease node 46 to node 15, causing violation of heap order. (b) Node 15 becomes root, and it's previous parent node 24 gets marked.

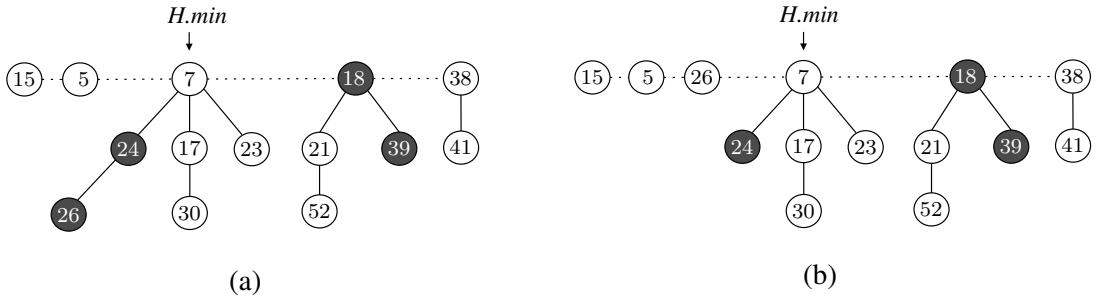


Figure 9.7: (a) Decrease node 35 to node 5 in Figure 9.9(b), causing violation of heap order. Node 5 becomes root. (b) It's previous parent node 26 was marked, so a cascading cut occurs and becomes root.

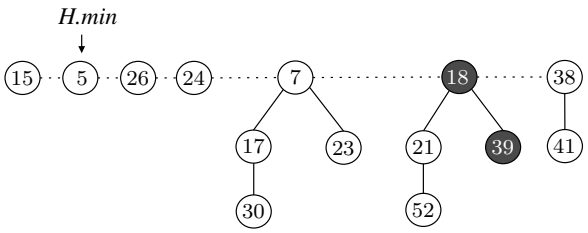


Figure 9.8: Node with key 26 had previous parent node 24 in Figure 9.10(b), it was marked, so a cascading cut occurs and becomes root.

As there's no more cascading cut, it stops!

9.4 Bounding the maximum degree

Define $size(x)$ to be the number of nodes, including x itself, in the sub-tree rooted at x. We shall show that $size(x)$ is exponential in degree of x, denoted $d(x)$.

Lemma 9.4.1. *Let x be a node of degree of k with children $y_1, y_2, \ldots, y_k$ which were linked to x in order from earliest to latest. Then, $d(y_1) \geq 0$ and $d(y_i) \geq i - 2$ for*

$i = 2, 3, \ldots, k$.

Proof. For $i = 1$, the case is trivial, that is $d(y_1) \geq 0$. For $i \geq 2$, we note that when y_i get linked to x, all of $y_1, y_2, \ldots, y_{i-1}$ were already children of x, so we must have had $d(x) \geq i - 1$. Because node y_i is linked to x by consolidation only if $d(x) = d(y)$, we must have also had $d(y_i) \geq i - 1$ at that time. Since then, node y_i may have lost at most one child, since if it had lost two children it would have been cut from x by cascading. Thus we conclude that $d(y_i) \geq i - 2$. □

Lemma 9.4.2. *Let F_k denote k-th Fibonacci number. Prove that for $k \geq 0$*

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i.$$

Proof. For $k = 0$, $1 + 0 = 1 = F_2$, the expression is true. Assume that for $k \geq 0$, $F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$. Then $F_{k+2} = F_k + F_{k+1} = F_k + \left(1 + \sum_{i=0}^{k-1} f_i\right) = 1 + \sum_{i=0}^{k} F_i$ □

Lemma 9.4.3. *For all integers $k \geq 0$,*

$$F_{k+2} \geq \phi^k,$$

where $\phi = \frac{1+\sqrt{5}}{2} = 1.61803\ldots$, is the golden ratio, the positive root of the equation $x^2 = x + 1$.

Proof. For $k = 0$, $F_2 = 1 \geq \phi^0 = 1$, the expression is true. Also for $k = 1$, $F_3 = 2 \geq \phi^1 = 1.61803$ the expression is true. Assume for $k \geq 2$, $F_{k+2} \geq \phi^k$. Then

$$\begin{aligned} F_{k+3} &= F_{k+1} + F_{k+2} \\ &\geq \phi^{k-1} + \phi^k \\ &= \phi^{k-1}(1 + \phi) \\ &= \phi^{k-1}\phi^2 \\ &= \phi^{k+1}. \end{aligned}$$

□

Lemma 9.4.4. *Let x be any node in a Fibonacci, and let $d(x) = k$. Then size$(x) \geq F_{k+2}$.*

Proof. Let $y_1, y_2, \ldots, y_k$ denote the children of x in the order in which they were

linked to x. Easy to see that

$$size(x) \geq 2 + \sum_{i=2}^{k} size(y_i),$$

where x contributes 1, y_1 contributes at least 1 as $d(y_1) \geq 0$. Let s_i denote the minimum size of any node having degree i. Then by Lemma 9.4.1

$$size(x) \geq s_k \geq 2 + \sum_{i=2}^{k} s_{i-2}$$

Next, we claim that $s_k \geq F_{k+2}$. For $k = 0, 1$ the claim is true. Assume that the claim holds for $k \geq 2$, then

$$s_{k+1} \geq 2 + \sum_{i=2}^{k+1} s_{i-2} \geq 2 + \sum_{i=2}^{k+1} F_i \geq 1 + \sum_{i=0}^{k+1} F_i.$$

By Lemma 9.4.2

$$s_{k+1} = F_{k+3}.$$

It completes the proof that $size(x) \geq F_{k+2}$. □

Theorem 9.4.5. *The maximum degree $D(n)$ of any node in an n-node Fibonacci heap is $O(\log n)$.*

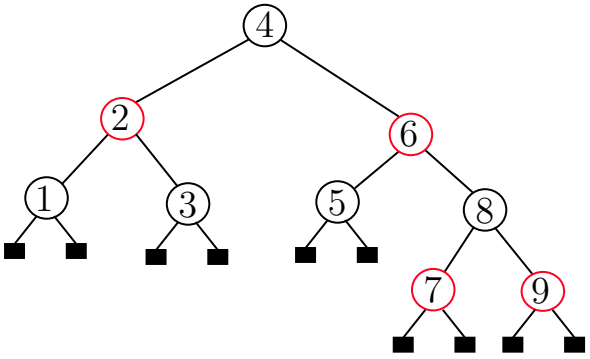
Proof. By Theorem 9.4.4, $n \geq size(D(n)) \geq F_{D(n)+2} \geq \phi^{D(n)}$. Taking logarithms gives $D(n) \leq \log_\phi n$. Thus the maximum degree $D(n)$ is $O(\log n)$. □

Chapter 10

Red Black Trees

A red black tree is a binary search tree where a node has given a color which can be either red or black. By constraining the node colors on any simple path from the root to a leaf, red black tree ensure that no such path is more than twice as long as any other, so the tree is approximately balanced. Formally, a red black tree is a binary search tree that satisfies the following properties:

1. The root is black.
2. Every leaf is black, we will consider them as NIL, they do not contain any value.
3. If a node is red, then both of its children are black.
4. For each node, all simple paths from the node to descendent leaves contains the name number of black nodes. The number of black nodes on any simple path from, but not including a node x down to a leaf is called the black heights of a node x, denoted $bh(x)$.



(Sometimes we will omit leaf nodes when drawing a red black tree.) We shall regard key bearing nodes as being internal nodes of the tree.

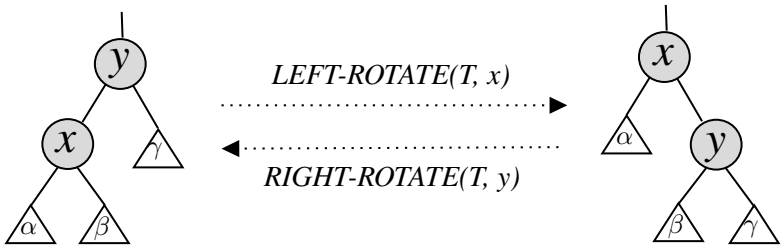
Lemma 10.0.1. *The subtree rooted at any node x contains at least $2^{bh(x)}$-1 internal nodes.*

Proof. We prove this claim by induction on the height of x. If the height of x is 0, then x must be leaf, the subtree rooted at x indeed contains $2^{bh(x)} - 1 = 1 - 1 = 0$ internal nodes. For the inductive step, consider a node x that has a positive height and is an internal node with two children. Each child has a black height of either $2^{bh(x)}$ or $2^{bh(x)} - 1$ depending on whether its color is red or black respectively. Since the height of child of x is less than the height of x itself, we can apply the inductive hypothesis to conclude that each child has at least $2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 1 = 2^{bh(x)} - 1$ internal nodes which proves the claim. □

Lemma 10.0.2. *A red black tree with n internal nodes has height at most $2\log(n+1)$.*

Proof. Let h be the height of tree. At least half the nodes on any simple path from the root to a leaf not including the root, must be black. Consequently, the black height of the root must be at least $\frac{h}{2}$. Thus by Lemma 10.0.1, $n \geq 2^{\frac{h}{2}} - 1 \Rightarrow h \leq 2\log(n+1)$. □

What are the consequences of the fact that $h \leq 2\log(n+1)$?
SEARCH, MINIMUM, MAXIMUM, SUCCESSOR and PREDECESSOR operations are in $\mathcal{O}(\log n)$ time.
How to do tree insert and tree delete operations?
Rotations:



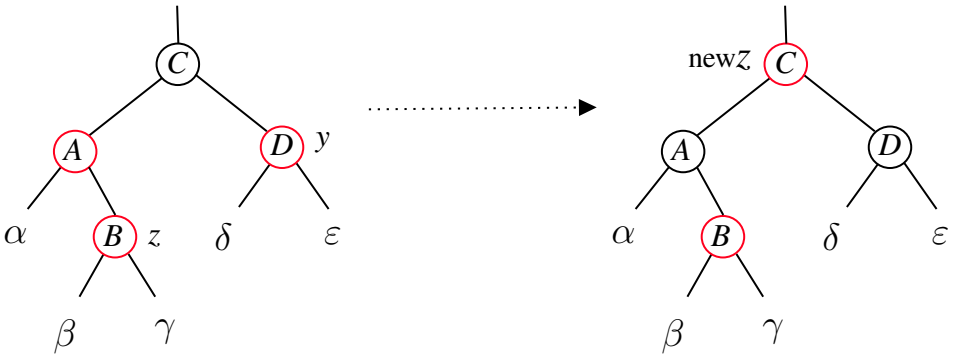
The letters α, β and γ represent arbitrary subtrees.

10.1 Insertion in Red Black Tree

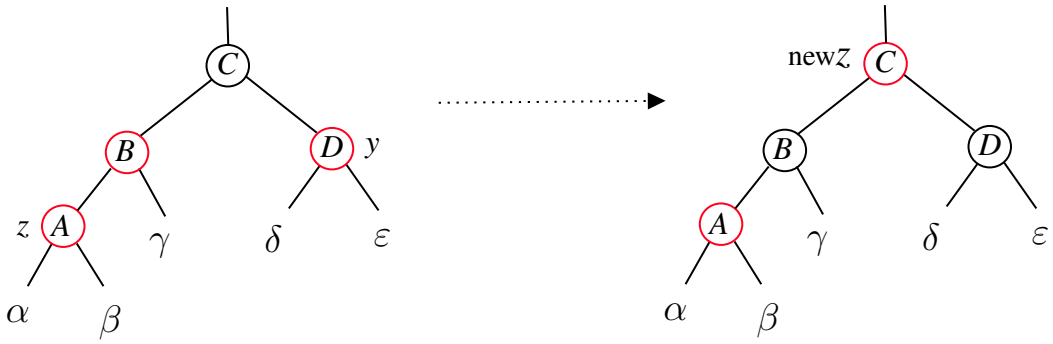
For inserting a node z in a red black tree, we have to consider the following three cases:

Case 1: *z's* uncle *y* is red

1. When z is a right child



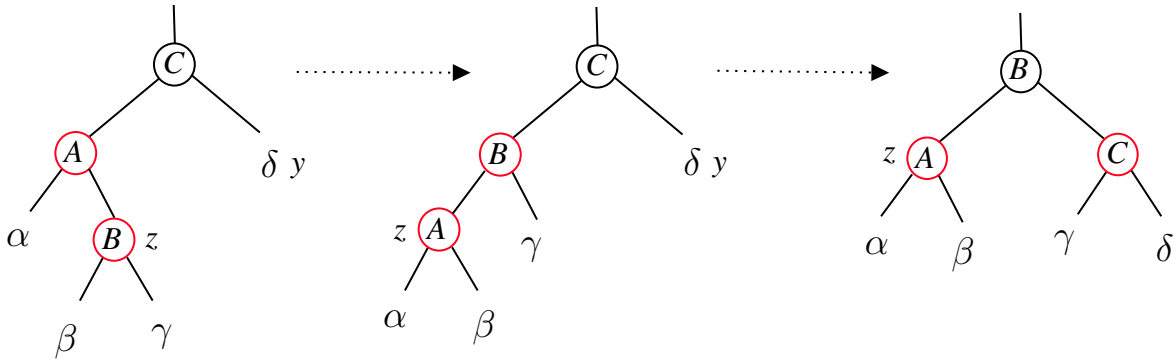
2. When z is a left child



In case 1 property 4 is violated, since z and its parent $z.p$ both are red. We take the same action whether **(1)** z is right child or **(2)** z is left child. Each of the subtrees $\alpha, \beta, \gamma, \delta$, and ε has black root, each has the same black height. Case 1 changes the color of some nodes, preserving property 5 i.e all downward simple paths from a node to a leaf have the same number of blacks. Any violation of property 4 can occur only between the new z, which is red, and its parent, if it is red as well.

Case 2: *z's* uncle *y* is black and *z* is a right child

Case 3: *z's* uncle *y* is black and *z* is a left child

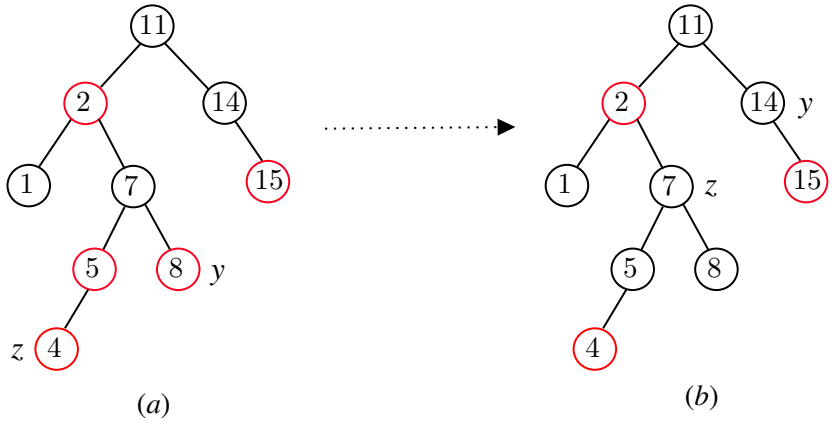


Case 2 Case 3

After right rotation and some color changes in above tree.

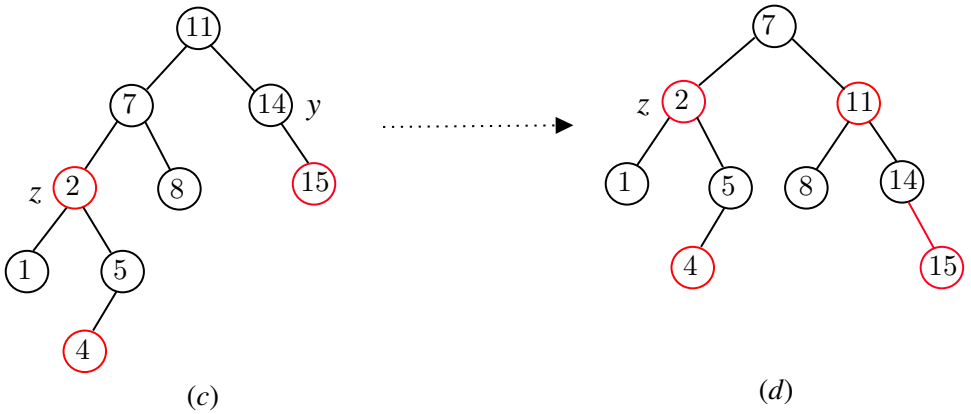
In cases 2 and 3, the color of z's uncle y is black. According to these two cases z would be a right child or left child of $z.p$. In case 2 z is the right child of its parent. Use the left rotation to transform the situation into case 3 in which node z is left child and now case 3 causes some color changes and a right rotation, which preserves the property 5 i.e all downward simple paths from a node to a leaf have the same number of blacks and the property 4 is also satisfied: there is no longer two red nodes in a row.

Example: Suppose is to be added in the following red black tree :



(*a*) (*b*)

In the above figure (a), A node z after insertion. Because both z and its parent $z.p$ are red, a violation of Property 4 occurs. Since $z's$ uncle y is red, case 1 applies in which we recolor nodes and move the pointer z up the tree, which is shown in the below figure(b).



(*c*) (*d*)

Once again, z and its parent are both red, but $z's$ uncle y is black. Since z is the right child of $z.p$, Case 2 applies in which we perform a left rotation, and the resultant tree is shown in below figure (c).

Now, z is the left child of its parent, and case 3 applies. So recoloring and right rotation yield the tree shown in above figure (d) which is legal red black tree.

Chapter 11

Splay Trees (Daniel Sleator and Robert Tarjan, 1985)

Some interesting facts about **Splay Trees** are:

1. Type of a binary search tree
2. Most of the operations have $O(\log n)$ time complexity, but some can be very slow as $O(n)$.
3. Not strictly balanced, (but it is faster)
4. Easy to implement
5. Most popular data structure in the industry
6. Fast access to elements accessed recently, for example caches
7. Splay tress are kept balanced with the help of rotations.

For splaying, we have to make rotations, that is, we update the references (pointers) which can be done in $O(1)$ time complexity.

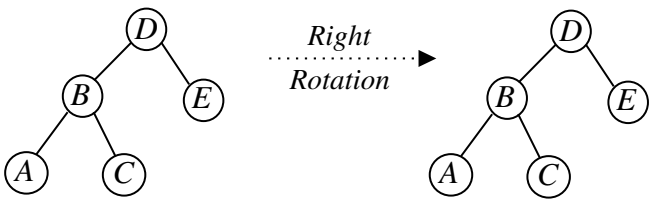


Figure 11.1: Single rotate operation on binary search tree.

11.1 Find Operation

It is like a standard BST search operation. After find the element we make rotations to make it roots, this process is called splaying. This is done because in the next searches it can be accessed very fast in $O(1)$ time. There are 3 ways we can make it happen:

1. zig-zag situation

2. zig-zig situation

3. zig situation

1. zig-zag situation: The given node x is a right child of a left child or the given node x is a left child of a right child.

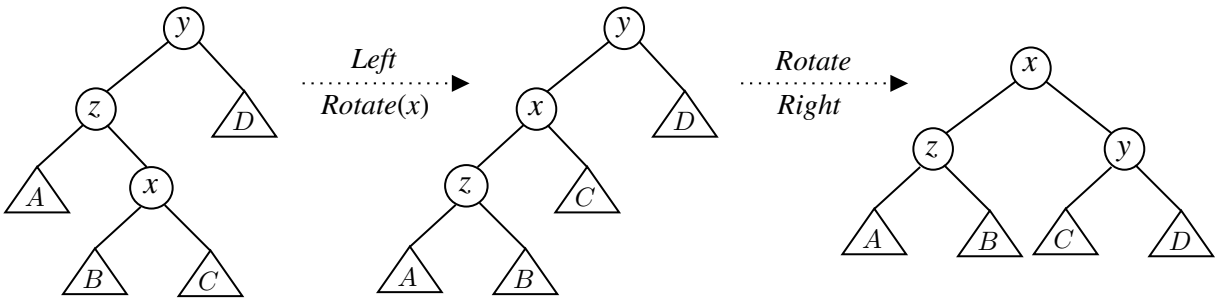


Figure 11.2: zig-zag step in splay operation

We care about making the tree balanced but do not care that much; it is important to make the search key the root.

(b) zig-zig situation: The given node x is a left child of a left child or the given node x is a right child of a right child. The following rotation is required.

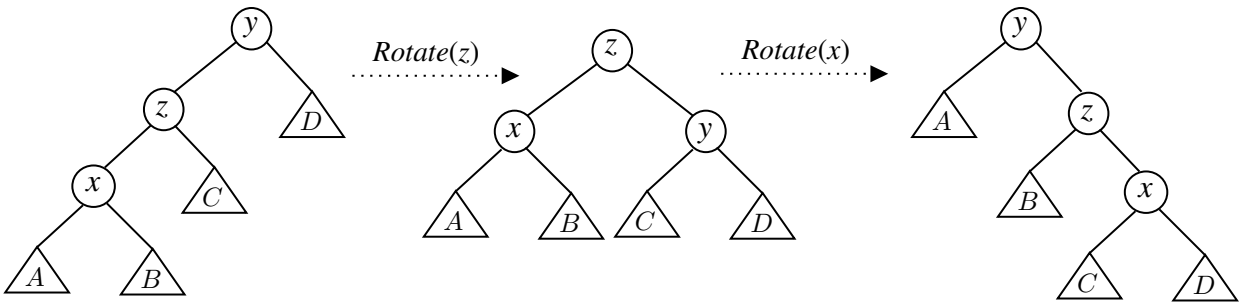
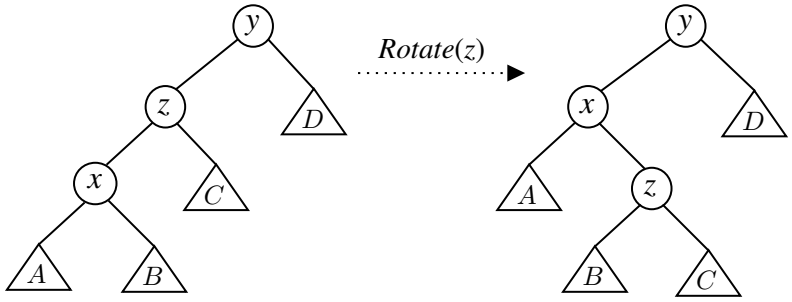
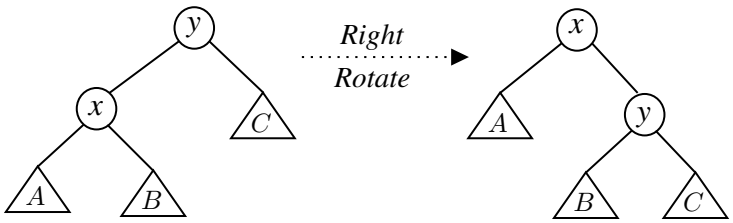


Figure 11.3: zig-zig step in splay operation

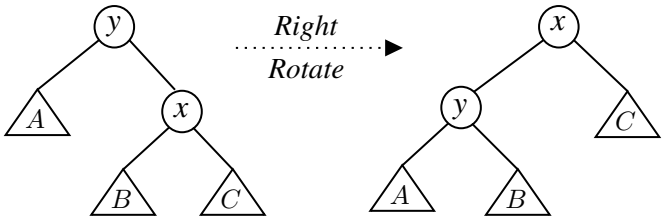
Wrong rotate node z on right:



(c) zig situation : We have to repeat the previous two steps over and over again until we get to the root. Sometimes we end up at the left/right child of the root: we just have to make a single right/left rotation accordingly. So here x is a child of the root. Just a single rotation is needed.



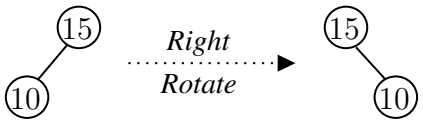
Or



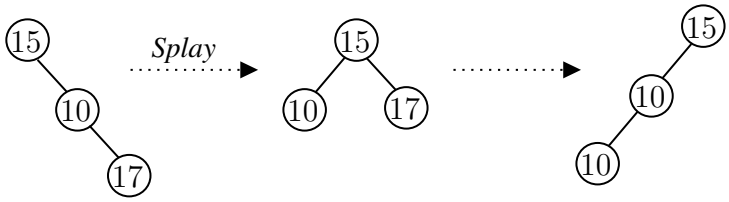
11.2 Insertion in splay tree

In the below example, we look how insertion happens in splay tree. Following are 6 elements to be sequentially inserted in a splay tree 15, 10, 17, 7. Inserting 15:

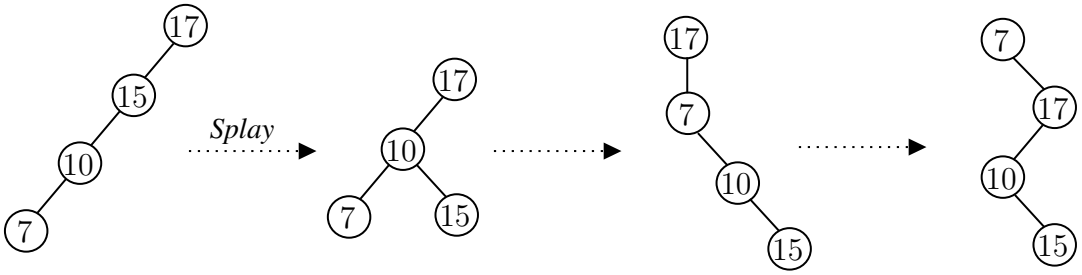
Inserting 10:



Inserting 17:



Inserting 7:



11.3 Deletion in splay tree

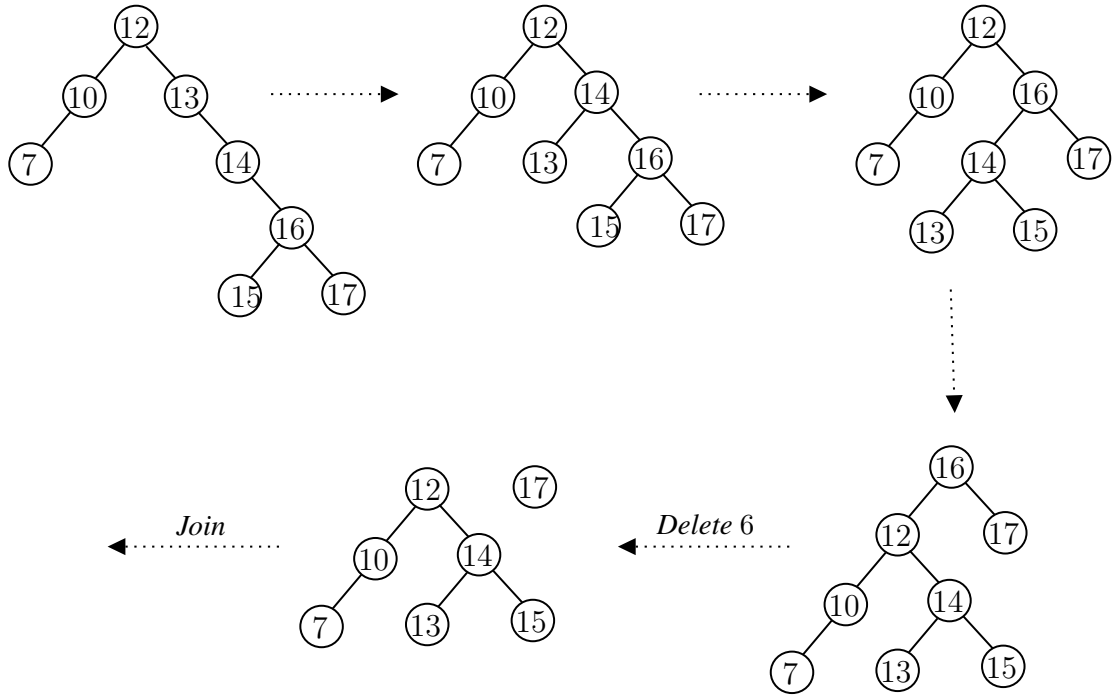
This can be done in two ways:

1. Top-down splaying
2. Bottom-up splaying

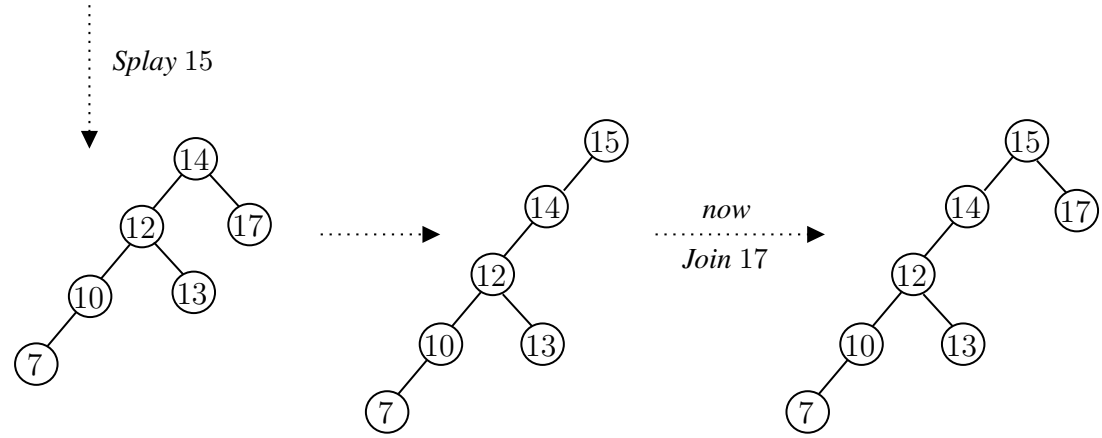
1. Top-down splaying: Here, to delete we do three main steps:

- make root
- delete
- join

In the below example we will delete following nodes : 16



Before joining find maximum element in the left tree that is 15, splay it.



2. Bottom-up splaying: Here, to delete we do two main steps:

- Delete the node like in binary search tree.
- make its parent the root

Exercise 11.3.1. *Design and explain the operation of a node deletion in a red-black tree.*

Chapter 12

van Emde Boas Data Structure

The goal is to maintain a dynamic set S, where S is a subset of a fix universe $U = \{0, 1, 2, 3, \ldots, u-1\}$

1. Dynamic Operations
    - Insert or delete an element $x \in U \setminus S$
2. Query Operation
    - The successor or the predecessor of an element $x \in U$

Example 12.0.1. *For $u = 16$, $U = \{0, 1, 2, \ldots, 16\}$. Let $S = \{1, 9, 10, 15\}$, then successor(1) = 9, successor(2) = 9*

We want to solve these operations very fast $O(\log \log u)$. So we need the recurrence relation $T(u) = T(\sqrt{u}) + O(1)$.

Exercise 12.0.2. *Solve $T(u) = T(\sqrt{u}) + O(1)$.*

12.0.1 Bit vector

In a bit vector for a dynamic set S the i-th entry is 1 if that entry is present in S, otherwise it is 0. For $S = \{1, 9, 10, 15\}$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	1

Insertion and deletion takes constant time (make 0, 1 as required). What about successor?

We break the bit vector in blocks/widgets each of size $\sqrt{u}$.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1

It can be visualised as a 2D array.

Widgets/Blocks

$w[0]$

	0	1	2	3
	0	1	0	0

$w[1]$

	0	1	2	3
	0	0	0	0

$w[2]$

	0	1	2	3
	0	1	1	0

$w[3]$

	0	1	2	3
	0	0	0	1

$S = \{1, 9, 10, 15\}$
Total number of blocks/widgets = $\sqrt{u}$
and size of each block/widget = $\sqrt{u}$

$w[i]$ =

0	1					$\sqrt{u}$-1
		-	-	-	-	-	

How to insertion and delete? The position of an element x can be find using the high value $H(x)$ and the low value $L(x)$ in the binary representation of x.

x =

$u/2$	$u/2$
$H(x)$	$L(x)$

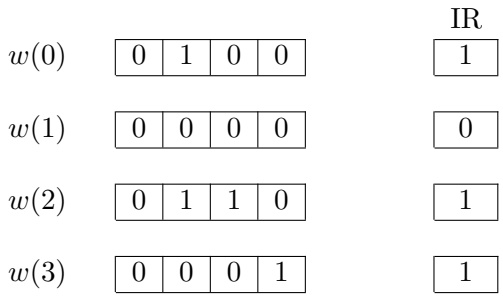
$x = H(x)\sqrt{u} + L(x)$
$H(x)$ = Widget Number
$L(x)$ = Position in the widget

$$x = 9 = \begin{array}{cc} 10 & 01 \\ H(x) & L(x) \\ = 2 & = 1 \end{array}$$

Insert(x)
$w(H(x), L(x)) = 1$.
Delete(x)
$w(H(x), L(x)) = 0$.

Is there any benefit of breaking a one dimensional array into two dimensional array. Successor query in worst case still takes $\sqrt{u} \times \sqrt{u} = O(u)$ time. How can we improve it? How to augment widget to avoid all zero block? Let us do it by adding an array

whose i-th entry tells whether widget $w(i)$ has all the entries zero or not. Let us call it an indicator array (IR).



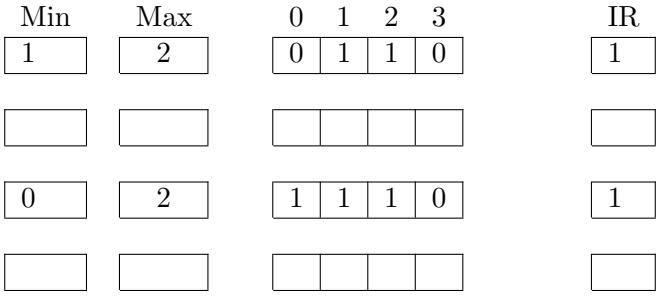
Successor$(x, w((), ()))$

1. Look at the successor of the x with in widget $w(H(x))$ starting after $L(x)$. It takes $O(\sqrt{u})$ time. If the successor is found return.
2. Else find the next nonempty block/widget $w(i)$. It will take order of $O(\sqrt{u})$.
3. Find the successor in $w(i)$ starting from beginning. It will take order of $O(\sqrt{u})$.

So there can be 3 recursive query in worst case. Thus the recursive expression is still

$$T(u) = 3T(\sqrt{u}) + O(1).$$

We need further augmentations by adding Min, Max arrays whose i-th entry tells the first, last index with 1 in the widget $w(i)$.



(Empty cells are zero)
Now it easy to check the successor query in just one recursive call.

1. if max $\geq L(x)$, search in the widget $w(H(x))$.
2. if max $\leq L(x)$, make the search query in the IR, the min corresponding to next widget with IR value 1 is the successor for x.

Now the recursion is

$$T(u) = T(\sqrt{u}) + \theta(1),$$

which makes the search querry in $O(\log \log n)$.

Chapter 13

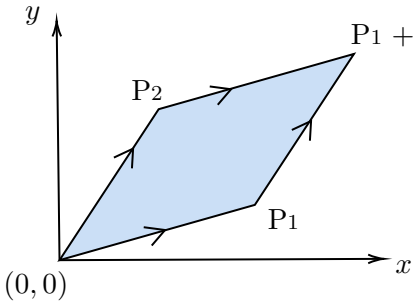
Computational Geometry

In this chapter we consider algorithms over points and line segments in 2D space. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be two distinct points. A convex combination of P_1 and P_2 is any point $P_3 = (x_3, y_3)$ such that for some α in range $0 \leq \alpha \leq 1$, we have $x_3 = \alpha x_1 + (1 - \alpha)x_2$ and $y_3 = \alpha y_1 + (1 - \alpha)y_2$. We also write that $P_3 = \alpha P_1 + (1 - \alpha)P_2$. Intuitively P_3 is any point that is on line passing through P_1 and P_2, the line segment $\overline{P_1P_2}$ is the set of convex combinations of P_1 and P_2 (we call P_1 and P_2 the end points of segment $\overline{P_1P_2}$). Sometimes the ordering of $P1$ and $P2$ matters, and we speak of the directed segment $\overrightarrow{P_1P_2}$. If P_1 is the origin (0, 0), then we treat the directed segment $\overrightarrow{P_1P_2}$ as vector P_2.

The first question that we are interested in is to determine whether given two line segments intersects or not? We will not use any division or trigonometry function to avoid the floating point errors. Fortunately cross product of vectors comes handy for various computational geometry algorithms.

Cross Product

We can interpret the cross product $P_1 \times P_2$ as the signed area of parallelogram formed by the points, (0, 0), P_1, P_2, $(P_1 + P_2)$



The area (with sign) of the parallelogram formed by the points, $(0, 0), P_1, P_2, (P_1 + P_2)$

is

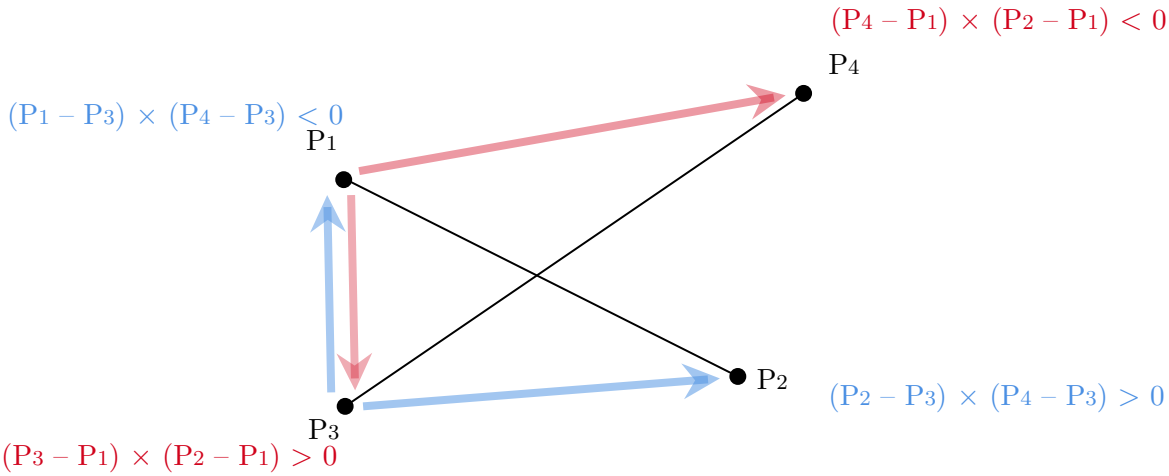
$$P_1 \times P_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1 y_2 - x_2 y_1$$
$$= -P_2 \times P_1$$

If $P_1 \times P_2$ is positive, then P_1 is clockwise from P_2 with respect to the origin $(0, 0)$.

- **Determining whether consecutive segments turn left or right**
  Given $\overline{P_0P_1}$, $\overline{P_1P_2}$, whether the left turn or the right turn is required at point P_1?
  Ans: If $(P_1 - P_0) \times (P_2 - P_0)$ is positive, P_1P_0 is clockwise from P_2P_0, we take left turn at P_1.

- **Determining whether two line segments intersects**
  To determine whether given two line segments intersect, we check if each segment straddles the line containing the other. A segment $\overline{P_1P_2}$ straddles a line if point P_1 lies on one side of the line and point P_2 lies on the other side. A boundary case arise if P_1 or P_2 lies directly on the line. Two line segments intersect if and only if either (or both) of the following conditions hold:

  1. Each segment straddles the lie containing the other.
  2. An endpoint of one segment lies on the other segment (boundary condition).



13.0.1 Determining whether any pair of segments intersect

Trivial way checks all $\binom{n}{2}$ pairs. Every pairs take constant time, which gives $O(n^2)$.

Sweep Line Method:

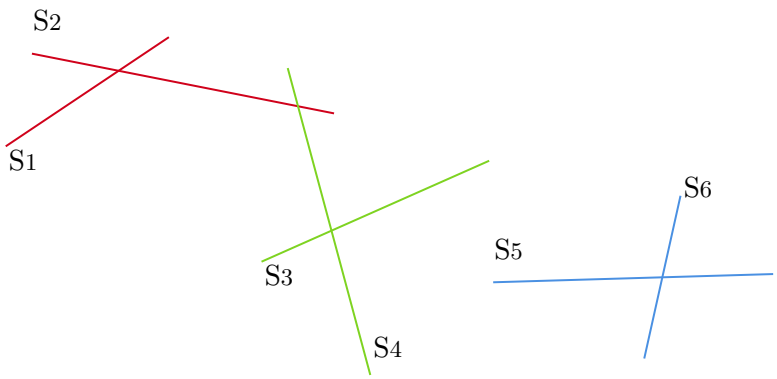
A vertical line that moves from left to right process the event points. The event points are the end points of line segments. The sweep line procedure is as follows.

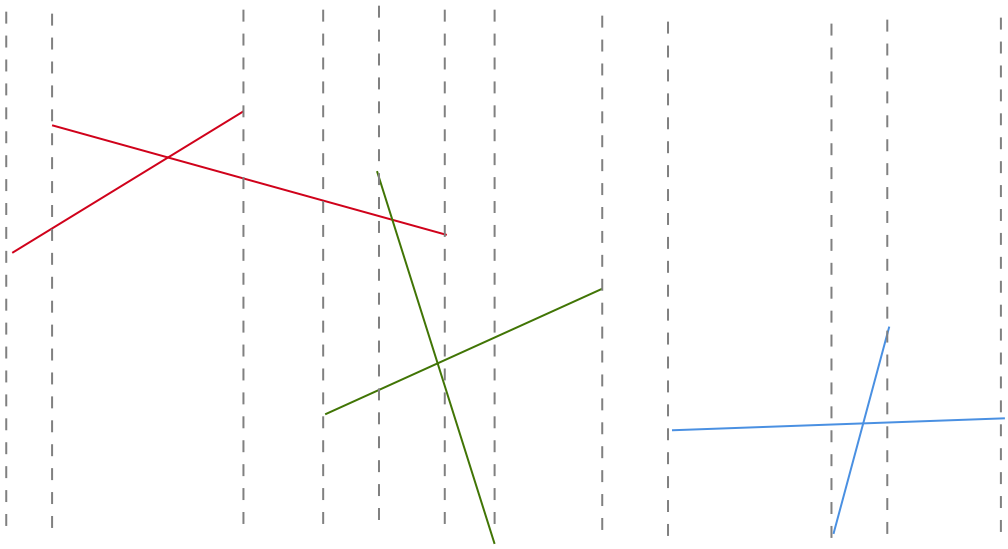
1. Sort the event points from left to right, breaking the ties by putting left endpoints before right endpoints and breaking further ties by putting points with lower y-coordinate first.

2. For a point p in the sorted list, if p is the left endpoint and if (1) there exists a line segment just above p and it intersects with the line segment with p as an endpoint, or (2) there exists a line segment just below p and it intersects with the line segment with p as an endpoint, then the procedure returns an intersection.

3. For a point p in the sorted list, if p is the right endpoint and if (1) there exists a line segment just above p and a line segment just below p and if both these line segment intersects, then the procedure returns an intersection.

4. Otherwise, there is no intersection.





13.1 Convex Hull

The convex hull of a set Q of points denoted by $\mathrm{CH}(Q)$, is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior.

13.1.1 Graham's scan

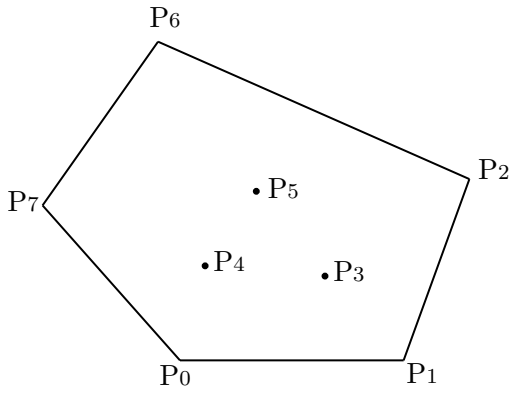
Graham's scan solves the convex-hull problem by maintaining a stack S of candidate points. Note that point p_0, the point with the lowest y-coordinate (or the leftmost such point in case of a tie) must always be in $\mathrm{CH}(Q)$. The rest of the procedure is as follows.

1. Let $p_1, p_2, \ldots, p_m$ be the points in Q sorted by polar angle in counterclockwise order around p_0. If more than one point has the same angle, then remove all but the one that is farthest from p_0.

2. We first push p_0, p_1, p_2 onto an empty stack S.

3. For $i = 3$ to m, check the angle formed by the points, just below the top of S, top of S, and p_i. If the angle makes a nonleft turn, remove the top point from S. Do this recursively. If the angle make left turn, push the point p_i onto the stack.

4. the points in S constitute the $\mathrm{CH}(Q)$.



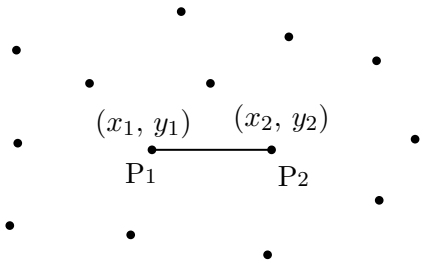
Jarvis's march

Jarvis's march builds a sequence $H = (P_0, P_1, \ldots, P_{n-1}$ of the vertices of CH(Q). We start with P_0. The next vertex P_1 in the convex hull has the smallest polar angle with respect to P_0 (in case of ties, we choose the point farthest from P_0). Similarly, P_2 has the smallest polar angle with respect to P_1, and so on. When we reach the highest vertex, say P_k, we have constructed the right chain of CH(Q). To construct the left chain, we start at P_k and choose P_{k+1} as the point with the smallest polar angle with respect to P_k, but from the negative x-axis. We continue so, forming the left chain by taking polar angles from the negative x-axis, untill we come back to our original vertex P_0.

Time complexity: $O(nh)$, where h is the number of vertices of CH(Q).

13.2 Finding the closest pair of points

Given n points in a 2D-plane our task to find a pair (P_1, P_2) with the minimum distance among all the pairs.

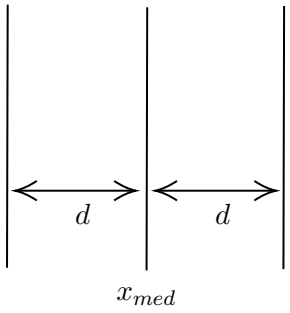


Naive way: For each of $\binom{n}{x}$ pairs of pints compute the distance and find the minimum. The complexity is $O(n^2)$.

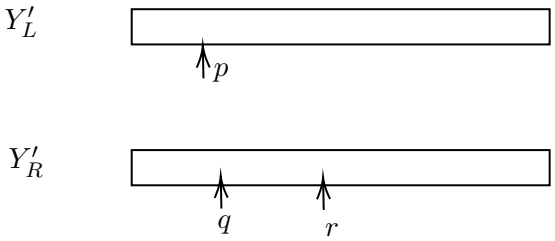
13.2.1 Divide and conquer approach:

Step I : Initialisation:
Sort the points by their x-coordinates, call the sorted set as X and by their y-coordinates, call the sorted set as Y. This step takes $O(n \log n)$ time.

Step II : Divide Step.
Let the value of median (when sorted on x-coordinate) be x_{med}.
- Split points into X_L and X_R points.
X_L have their x-coordinate $\leq x_{med}$. X_R contains the other points. (Both are sorted.)
- Similarly split Y into Y_L and Y_R points.
Y_L contains the same as X_L, sorted. Y_R contains the same as X_R, sorted.
In short, the input points are divided into X_L, Y_L and X_R, Y_R.
Step III : Recursive Step. Let
d_L : distance of the closest pair on input (X_L, Y_L).
d_R : distance of the closest pair on input (X_R, Y_R).
$d = \min(d_L, d_R)$.



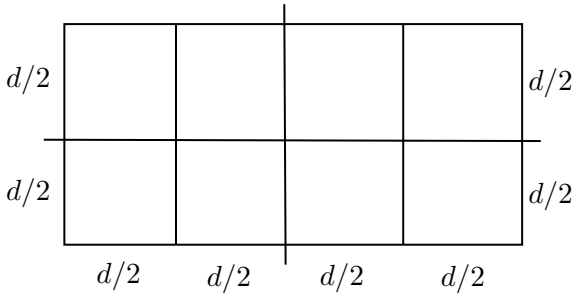
Step IV : Putting the answer together.
X'_L : Points from X_L whose x-coordinates are greater than $x_{med} - d$.
Y'_L : y-coordinates of the same set of points as in X'_L, sorted (pruning).
X'_R : Points from X_R whose x-coordinates are lesser than $x_{med} + d$.
Y'_R : y-coordinates of the same set of points as in X'_R, sorted (pruning).

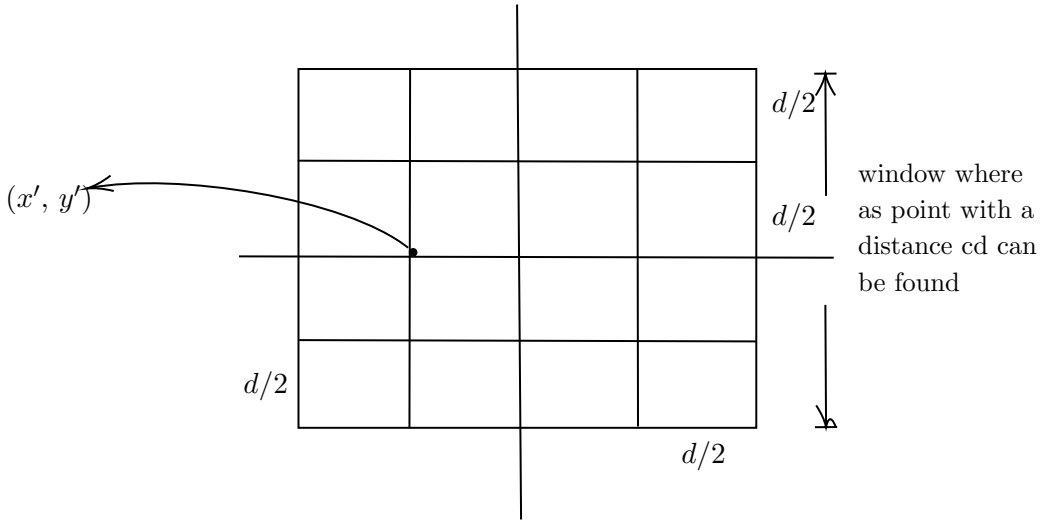


For each point in $Y'_L \cup Y'_R$ (in the sorted order), check its distance with at most 7 next immediate points. If the minimum among all these distance is less than d, we are done with this recursive step.
Other than the initialisation step, the time complexity is given by the following recurrence.

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

Hence the total time taken is $O(n \log n)$.





Chapter 14

P, NP, NP-Hard, NP-Complete

Given a string of a language, and a machine how much time the machine will take to execute/process this string? On what factors it depends?
Recall the complexity notations:

1. $O(n^c)$: n is size of input, c is some constant. This is called *polynomial time complexity.*

2. $O(c^n)$: exponential time complexity. Which complexity is desirable?

The machine we consider now is turning machine or our modern computers (Laptops, Workstation, etc.). They all are equivalent in their computation powers up to some constant.
For example for a string of length n, one computer may take $2n^2$ operators, while another computer may take $5n^2$ operations. But the complexity in both the cases is $O(n^2)$.
We now classify problems based on their time complexity. We will mainly focus on decision problems, the problems for which the answer is yes or no. Sometimes you may seems examples where problems are not decision problems, but we can always have their corresponding decision problem.

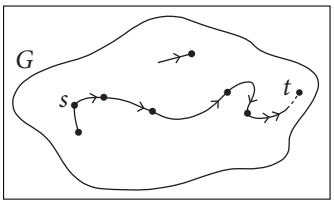
14.1 Class P

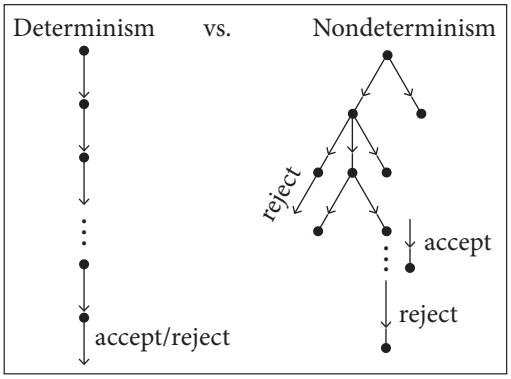
P is the set of problems those can be deterministically solved in polynomial time on turning machine (our computer).
Examples:

1. Sorting n numbers: $O(n \log n), O(n^2)$.

2. Matrix multiplication of two $n \times n$ matrices: $O(n^3)$.

3. Given a directed graph containing nodes s, t. Determine whether a directed path exists from s to t.





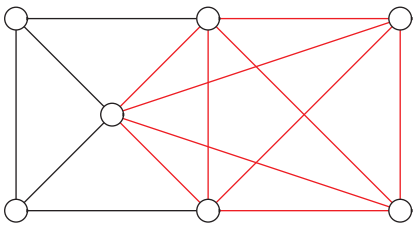
14.2 The Class NP

NP is the set of problems solved in polynomial time on a nondeterministic Turing machine.

or, equivalently

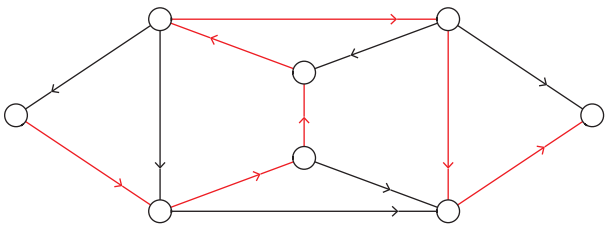
The set of problems that have polynomial time verifiers.

Example:



(1) Given an undirected graph find a k-clique inside it.

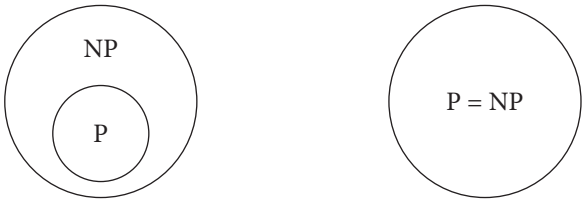
(2)



Finding a Hamiltonian path from s to t.

For both of these problems, if there is a certificate for a solution, we can check it in polynomial time. For example, in example (1) if we are given a subset of k vertices, we can check in polynomial time whether the subset forms a clique or not.

Since if a problem can be solved in a polynomial time then it can be verifiable easily, so $P \subset NP$. But is the converse true? Its a million dollar question.



One of these two possibilities is correct.

Polynomial time reducibility:

Problem A is polynomial time reducible to problem B, written $A \leq_p B$, if there exists a polynomial time algorithm (or function) $f : \varepsilon^* \to \varepsilon^*$, where for any input, w,

$$w \in A \Longleftrightarrow f(w) \in B.$$

Theorem: If $A \leq_p B$ and $B \in P$, then $A \in P$.

Let x be a Boolean variable (True/False). A literal is a Boolean variable or a negative Boolean variable, (x or $\bar{x}$). A clause is several literals connected with (or) $\vee$s, as in $(x_2 \vee \bar{x}_2 \vee \bar{x}_3 \vee x_4)$. A Boolean formula is in conjunctive normal form, called a cnf-formula, if it comprises several clauses connected with $\cap$s, as in

$$(x_1 \vee \bar{x}_2 \vee \bar{x}_3 \vee \bar{x_4}) \wedge (x_3 \vee \bar{x}_5 \vee x_6 \wedge \bar{x}_6).$$

It is a 3cnf-formula (3-SAT) if all the clauses have three literals, as in

$$(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (x_2 \vee \bar{x}_5 \vee x_6) \wedge (x_3 \vee \bar{x}_6 \vee x_4) \wedge (x_4 \vee \bar{x}_6).$$

In a breakthrough, Cook and Levin (1971) proved that all the problems in NP are reducible to the 3-SAT problem in polynomial time.

NP-Hard Problems:

The next important class of the problems are NP-hard problems. Let NP $= \{p_1, p_2, p_3, \dots\}$ be the set of NP problems. A problem X is called NP-hard if every problem in NP is

polynomial time reducible to X. That is $p_i \subseteq_p X$ for $i = 1, 2, 3, \ldots$. As stated earlier, 3-SAT is NP-hard.
Moreover, 3-SAT is also in NP, as we can always give a verifier.

NP-Complete

A problem X is called NP-complete if it is NP as well as a NP-hard problem.
Theorem: If B is NP-complete and $B \subseteq_p C$ for C in NP, then C is NP-complete.

14.3 Clique problem is in NP-complete

Clique Problem is to determine whether a graph contains a k-clique (complete graph on k nodes). It is easy to see that clique problem is in NP. We will now prove that is NP-hard, showing that it is NP-complete.

Theorem 14.3.1. *3-SAT is polynomial time reducible to clique problem.*

Proof. Let ϕ be a Boolean formula having k clauses such as

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k).$$

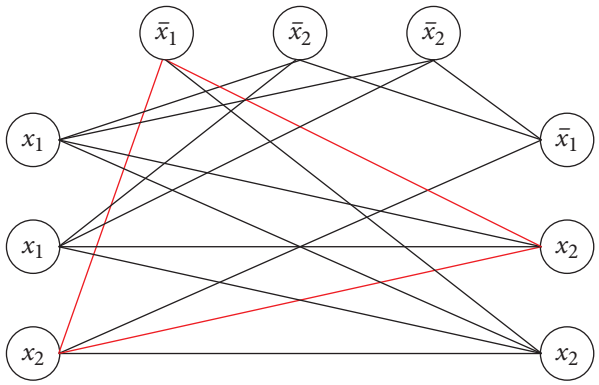
Using ϕ we construct a graph as follows.
The nodes in G are divided into k groups of three nodes each called the triples, $t_1, \ldots, t_k$. Each triple corresponds to one of the clauses in ϕ, and each node in a triple corresponds to a literal in the corresponding clause. Label each node of G with its corresponding literal in ϕ.
The edges of G connects all but two types of pairs of nodes in G. No edge is present between nodes in the same triple and no edge is present betwenn two nodes with contradictory lables, like x and $\bar{x}$.
Now we demonstrate why this construction works. We show that ϕ is satisfiable if and only if G has a k-clique.
Suppose that ϕ has a satisfying assignment. In that satisfying assignment, at least one literal is true in every clause. In each triple of G, we select one node corresponding to a true literal in the satisfying assignment. If more than one literal is true in a particular clause, we choose one of the true literals arbitrarily. The nodes just selected form a k-cliques. The number of selected is k, because we chose one for each of the k triples. Each pair of selected nodes is joined by an edge because no pair fits one of the exceptions described previously. They could not have contradictory labels beacause the associated literals were both true in the satisfying assignment. They could

not be from the same triple because we selected only one node per triple. Therefore G contains a k-clique.

Suppose that G has a k-clique. No two of the clique's nodes occur in the same triple because nodes in the same triple are not connected by edges. Therefore each if the k triples contains exactly one of the k cliques nodes. We assign truth values to the variables of ϕ so that each literal labeling a clique node is made true. Doing so is always possible because two nodes labeled in a contradictory way are not connected by an edge and hence both cannot be in the clique. This assignment to the variables satisfies ϕ because each triples contains a cliques node and hence each clause contains a literal that is assigned true. Therefore ϕ is satisfiable. □

Example Reduce $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2 \vee x_2)$ to a clique problem.

$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{v}_2 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2 \vee x_2).$$



$\bar{x}_1 = 1, x_2 = 1$.

The vertex cover problem: Let G be an undirected graph. A vertex cover of G is a subset of the nodes where every edge of G touches at least one of those nodes. The vertex cover problem asks whether a graph contains a vertex cover of a specified size.

Theorem 14.3.2. *The vertex cover problem is NP-complete.*

Proof Hint: Reduce clique problem to vertex cover problem. A graph G on n vertices has a clique of size k if and only if its complement graph $\bar{G}$ has a vertex cover of size $n - k$.

Chapter 15

Approximation Algorithms

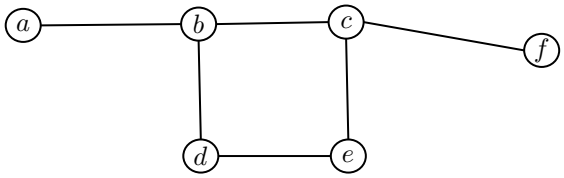
We have seen that the vertex cover problem is NP-complete. The optimization version of a vertex cover problem asks for a minimum vertex cover of a graph. So let us try to find a polynomial time approximation algorithm for a minimum vertex cover of a graph.
The following algorithm will give a vertex-cover in a graph G. **ALGO-1**:

1. Repeat the following until all edges in G touch a marked edge:

    a. Find an edge in G untouched by any marked edge.

    b. Mark that edge.

2. Output all nodes that are endpoints of marked edges.



Approximated vertex cover is $\{b, c, d, e\}$

Theorem 15.0.1. *ALGO-1 produces a vertex cover of G that is no more than twice as large as a smallest vertex cover.*

Proof. It obviously runs in polynomial time. Let X be the set of nodes that it outputs. Let H be the set of edges that it marks. We know that X is a vertex cover because H contains or touches every edge in G, and hence X touches all edges in G. Two facts:

1. $|X| = 2|H|$

2. $|H| \leq |Y|$, for any smallest vertex-cover Y.

   Every edge in H contributes two nodes to X, so $|X| = 2|H|$. Y is a vertex cover so every edge in H is touched by some node in Y. No such node touches two edges in H because the edges in H do not touch each other. Hence, X is no more than twice as large as Y. □

Minimum vertex cover problem is an example of a minimization problem. An approximation algorithm for a minimization problem is k-optimal if it always finds a solution that is not more than k times optimal. ALGO-1 is 2-optimal for minimum vertex-cover problem.
Now we will take a maximization problem known as max-cut problem. For a maximization problem, a k-optimal approximation algorithm fins a solution that is atleast $1/k$ times the size of the optimal.
Max-cut problem: A cut in an undirected graph is a separation of the vertices V into two disjoint subsets S and T. A cut edge is an edge that goes between a node in S and a node in T. The size of a cut is the number of cut-edges. the max-cut problem asks for a largest cut in the graph.
The following algorithm gives a approximate cut in G.
ALGO-2:

1. Let $S = \phi$ and $T = V$.
2. If moving a single node, either from S to T, or from T to S, increase the size of the cut, make that move and repeat this stage.
3. If no such node exists, output the current cut and halt.

Theorem 15.0.2. *Prove that ALGO-2 is 2-optimal.*

Proof. ALGO-2 runs in polynomial time because every execution of stage 2 increases the size of the cut to a maximum of the total number of edges of G. Now we show that cut by ALGO-2 is at least half the optimal cut. In fact we show ALGO-2 gives a cut containing half the edges of G. Observe that at every node of G, the number of cut edges is at least as large as the number of uncut edges, otherwise ALGO-2 would have shifted that node to the other side. We add up the number of cut edges at each node. The sum is twice the number of cut edges because every cut edge is counted once for each of it's two end points. By the preceding observation, that sum must be at least the corresponding sum of the numbered of uncut edges at every node. Thus G, has at least as many cut edges as uncut edges. Therefore, the cut contains at least half of all edges. □

15.1 Inapproximality

Unfortunately for some problems there is no approximation algorithm unless $P = NP$. One such problem is travelling salesman problem, defined as follows. Let G be a complete graph, where edges have nonnegative weights. The travelling salesman problem ask for a Hamiltonian cycle of minimum weight.

Theorem 15.1.1. *If $P = NP$, then for any constant $\rho \geq 1$, there is no polynomial-time ρ-approximation algorithm for travelling salesman problem.*

Proof. The proof is by contradiction. We will first reduce the Hamiltonian cycle problem (an NP-complete problem) to travelling salesman problem as follows. Let $G = (V, E)$ be an instance of the Hamiltonian-cycle problem. We turn G into an instance of the traveling-salesman problem as follows. Let $G' = (V, E')$ be the complete graph on V, that is,

$$E' = \{(u, v) : u, v \in V, u \neq v\}.$$

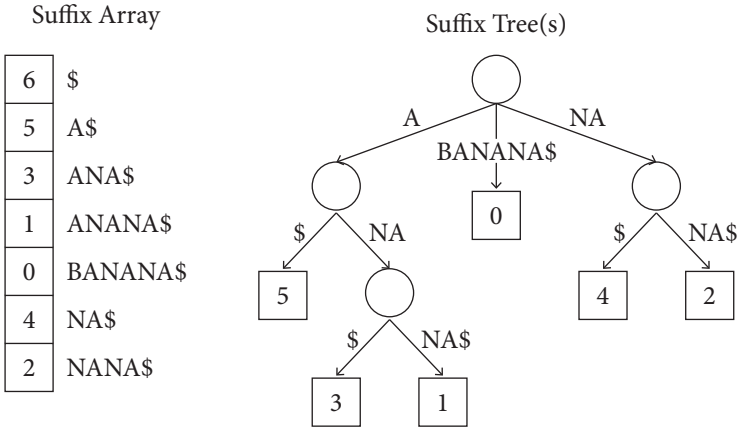
To each edge $(u, v) \in E'$ assign an integer cost $c(u, v) = 1$ if $(u, v) \in E$, otherwise, $c(u, v) = \rho|V| + 1$.
We can create representations of G' and c from a representation of G in time polynomial in $|V|$ and $|E|$.
If the original graph G has a Hamiltonian cycle H, then the cost function c assigns to edge of H a cost of 1, and so (G', c) contains a tour of cost $|V|$. On the other hand, if G does not contain a Hamiltonian cycle, then any tour of G' must use some edge not in E. But any tour that uses an edge not in E has a cost of at least

$$(\rho|V| + 1) + (|V| - 1) = \rho|V| + |V| > \rho|V|.$$

□

Chapter 16

Suffix-tries



Build a table containing all suffixes of a text T having m length. For example

$$
\begin{aligned}
T\text{:}\quad & \mathrm{G\,T\,T\,A\,T} \\
& \mathrm{G\,T\,T\,A\,T}\uparrow \\
& \quad \mathrm{T\,T\,A\,T} \\
& \qquad \mathrm{T\,A\,T} \quad m(m+1)/2 \\
& \qquad\quad \mathrm{A\,T} \\
& \qquad\qquad \mathrm{T}\downarrow
\end{aligned}
$$

First add special terminal character \$ to the end of T. \$ is a character that does not appear elsewhere in T it guarantees that no suffix is a prefix of any other suffix.

$$
\begin{aligned}
\mathrm{T:}\quad & \mathrm{G\,T\,T\,A\,T\,\$} \\
& \quad \mathrm{G\,T\,T\,A\,T\,\$} \\
& \qquad \mathrm{T\,T\,A\,T\,\$} \\
& \qquad\quad \mathrm{T\,A\,T\,\$} \\
& \qquad\qquad \mathrm{A\,T\,\$} \\
& \qquad\qquad\quad \mathrm{T\,\$}
\end{aligned}
$$

Suffix trie is a smallest tree such that:

1. Each edge is labeled with a character from Σ.
2. A node has at most one outgoing edge labeled with c, for any $c \in \Sigma$.
3. Each is "spelled out" along some path starting at the root.

Example: T: abaaba, $T\$$: abaaba\$

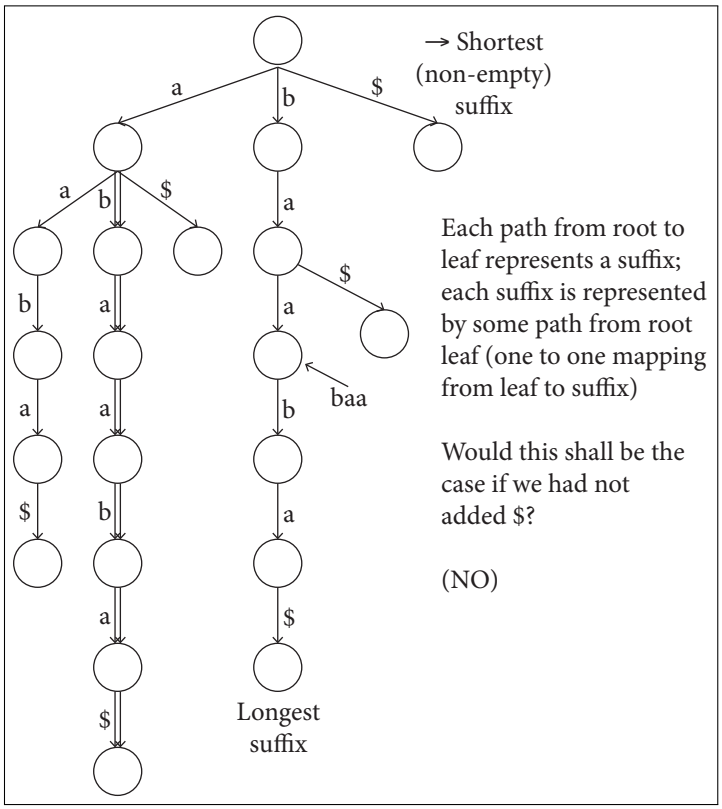
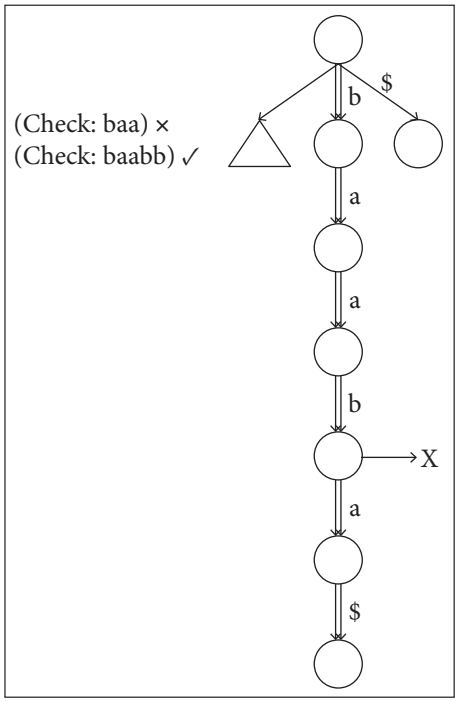


Figure 16.1: The longest suffix correspond to the path just left to where it is written 'longest suffix'.

We can think of nodes as having labels, where the label spells out characters on the path from the root to the node. How do we check with a string S is a substring of T? Note that each of T's substrings is spelled out along a path from the root, i.e. every substring is a prefix of some suffix of T.
Start at the root and follow the edges with the character of S. If we "fall off" the trie, that is, there is no outgoing edge for next character of S, then S is not substring of T. If we exhaust S without falling off, S is substring of T.



How do we check with a string S is a suffix of T? Same procedure as for substring, but additionally check whether the final node in the walk has an outgoing edge labeled S.
(Check: baa) ×
(Check: aba) ✓
How do we count the number of times a string S occurs as a substring of T? Follow path corresponding to S. Either we fall off, in which ease answer is 0, or we end up at node n and the answer = # of leaf nodes in the subtree rooted at n. Leaves can be counted with depth-first traversal.
How do we find the longest repeated substring of T? Find the deepest node with more than one child.
(Check: aba)

16.0.1 Space taken by suffix trie

How many nodes does the suffix trie have? Is there a class of string where the number of suffix trie nodes grows linearly with m?
Yes: e.g., a string of m a's in a row (a^m), T = aaaa

- 1 root
- m nodes with incoming 'a' edge.

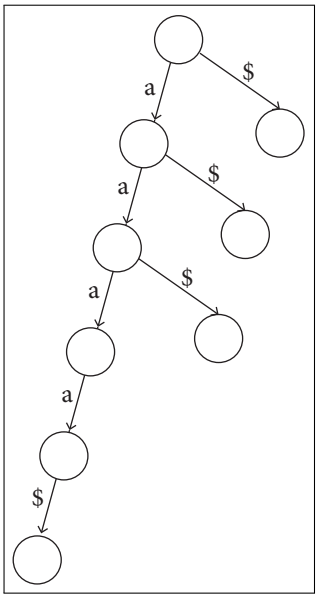


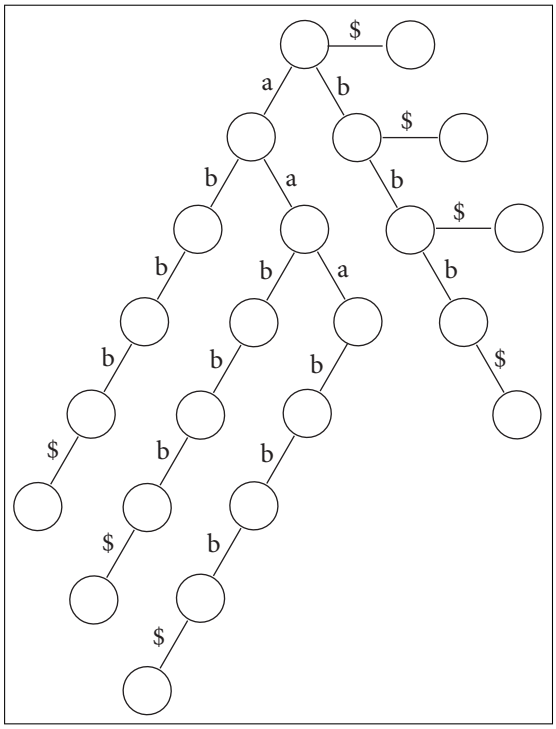
Figure 16.2: One $ is missing. (Corresponding to third last).

- $m + 1$ nodes incoming $ edge $2 + 2m$ nodes.

Is there a class of string where the number of suffix trie nodes grows with m^2?

Let T = aaabb, that is, of type $a^n bn$.

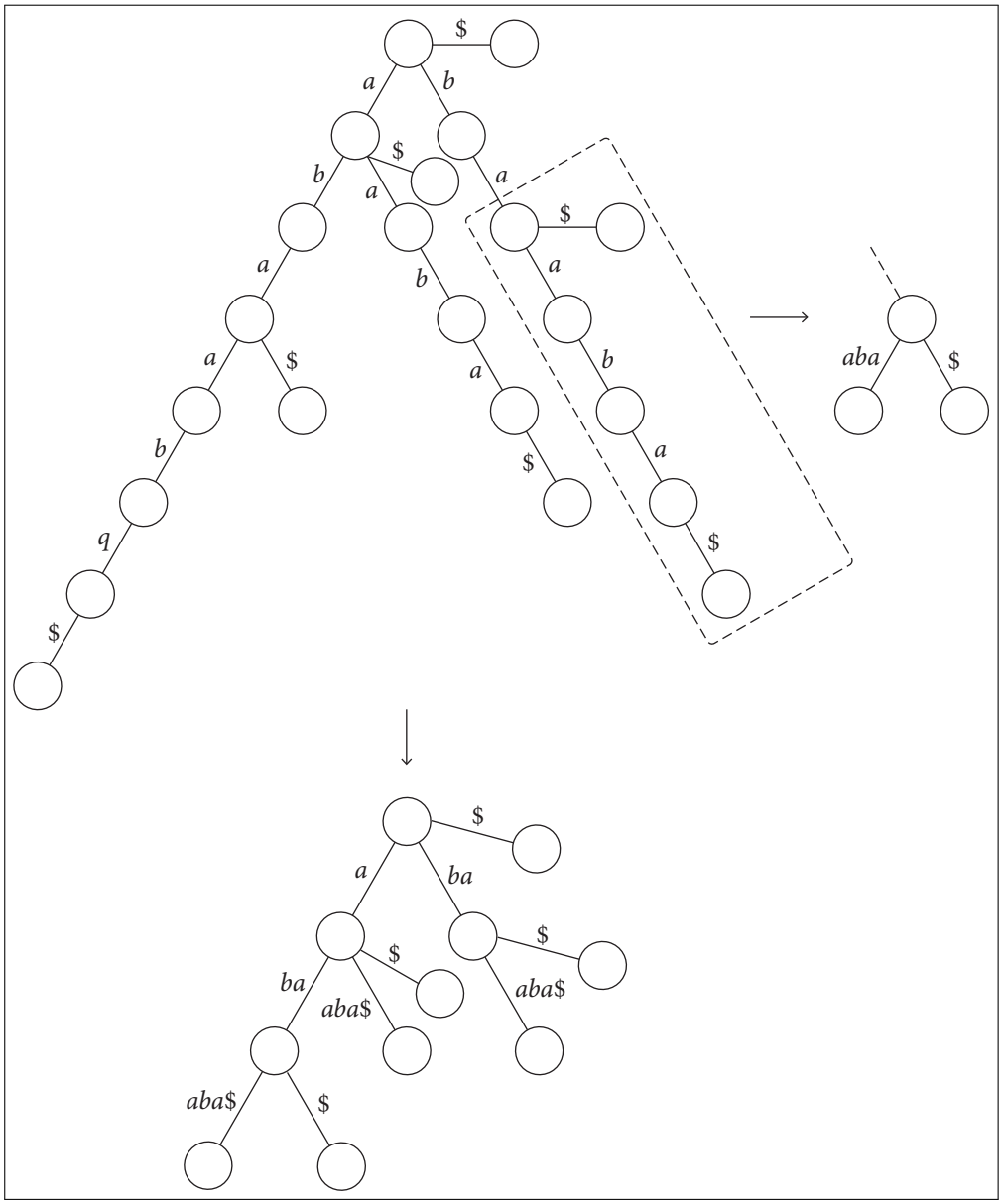
- 1 root.
- n nodes along "b chain", right.
- n nodes along "a chain", in the middle.
- n chain of n "b" nodes having off each "a chain" node.
- $2n + 1$ $ leaves.



Total nodes: $n^2 + 4n + 2$ nodes, where $m = 2n$.
In worst case the space requirement is of $O(m^2)$.

Suffix tree

How to reduce the space requirement. Again consider the string T = abaaba$



Idea 1: Coalesce non-branching paths into a single edge with a string label.

Reduces # nodes, edges guarantee internal nodes have > 1 child.

With respect to m :

How may leaves? $m + 1$

How many non-leaf leaf nodes? $\leq m - 1$.

(Hint. Compare with binary tree)

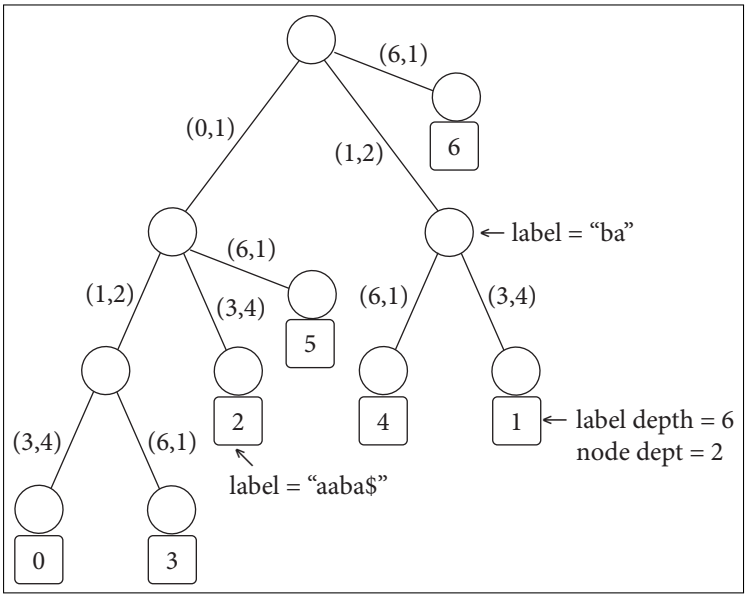
$\leq 2m - 1$ nodes in total, $O(m)$ nodes.

Is total size $O(m)$ now? No.

T = abaaba\$

Idea 2: Store T itself in addition to the tree. Convert tree's edge lebels (offset, length)

pairs with respect to T.



Now total data (space required for suffix tree) is $O(m)$.

Again, each node's lebel equals the concatenated edges labels from the root to the node. These are not stored explicitly.

Because edges can have a string labels we must distinguish two notions of "depth".

- Node depth: how many edges we must follow from the root to reach the node.
- Label depth: total length of edge labels for edges on path from root to node.
- Suffix tree: building.

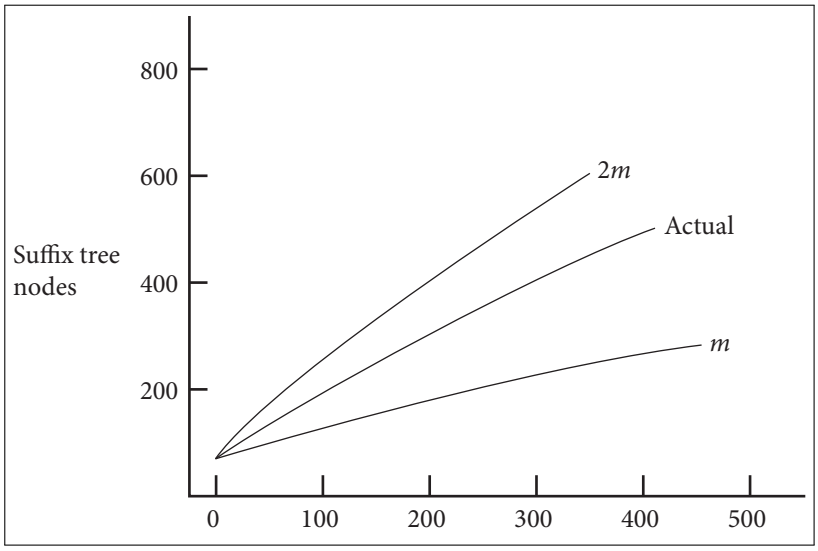
Naive method 1: build a suffix trie, then coalesce non-branching paths and relabel edges. $O(m^2)$ space, worst case $O(m^2)$ time.

Naive method 2: build a single-edge tree representing only the longest suffix, then augment to include the 2nd-longest, then augment to include 3rd-longest, etc.

$O(m)$ space.

(m^2) time.

Naive method 2 is described in Gusfield 5.4.



Best method: Ukkonen;s algorithm

$O(m)$ time and space!

Has outline propertyL: If T'arrives one character at a time, Ukkonen's algorithm efficiently updates suffix tree upon each arrival.

We wont cover it here, see Gusfield Ch. 6 for details.

Suffix tree:

How do we check with a string S is a substring of T?

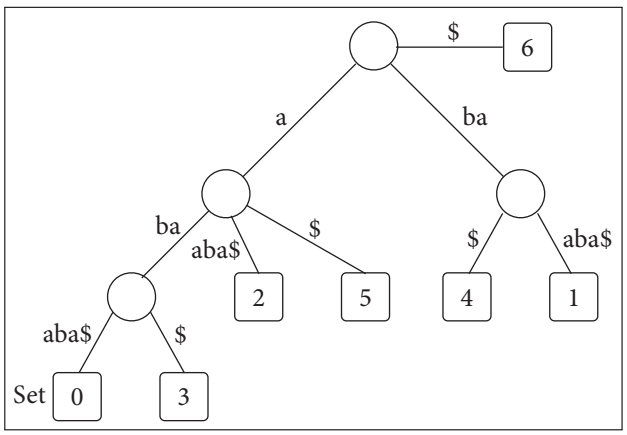
Essentially same procedure as for suffix trie, except we have to deal with coalesced edges.

Similarly for suffixes.

Applications

With suffix tree of T, we can find all matches of P to T. Let $k = \#$ makes.

E.g. P = ab, T = abaaba$.



Step 1: Walk down ab path $O(n)$. If we "fall off" there are no matches.

Step 2: Visit all leaf nodes below report each leaf offset as match off $O(k)$ time?

Total $O(n + k)$ time.

Find *the lowest common substring.*

To find the longest common substring (LCS) of X and Y, make a new string X # Y$ where # and $ are both terminal symbols. Build a suffix tree for X#Y$.