1

# Algorithms II

Sudebkumar Prasant Pal
email:spp@cse.iitkgp.ernet.in
Department of Computer Science and Engineering and
Centre for Theoretical Studies
Indian Institute of Technology, Kharagpur 721302, India.

August 25, 2019

## Contents



*Unfinished version. Only to be used as class notes. Not for circulation. Based mostly on the textbook "Introduction to Algorithms", Second Edition, by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, and also other textbooks.

# 1 The LCS problem

For sequences $X = x_1, x_2, \cdots, x_m$ and $Y = y_1, y_2, \cdots, y_n$, let $Z = z_1, z_2, ..., z_k$ be any LCS of $X$ and $Y$. Suppose $x_m = y_n$. Then we can find an LCS $Z'$ of $X_{m-1} = x_1, x_2, \cdots, x_{m-1}$ and $Y_{n-1} = y_1, y_2, \cdots, y_{n-1}$, and append $x_m = y_n$ to get a *common subsequence* of $X$ and $Y$. Is $Z'||x_m$ an LCS of $X$ and $Y$, of length $k$? For the sake of contradiction, suppose $Z'||x_m$ is not an LCS of $X$ and $Y$. The length of $Z'$ cannot be lesser than $k-1$ because the LCS $Z$ of $X$ and $Y$ is of length $k$ and $Z'||x_m$ would then be a common subsequence of $X$ and $Y$ of length less than $k$, a contradiction. So, the length of $Z'$ must be at least $k-1$. If it is $k-1$ then we are done. Otherwise, $Z'$ has length $k$ or more. Is this possible at all? The following Theorem 1 comes to our rescue. It says that $Z_k = Z$ being an LCS of length $k$ for $X = X_m$ and $Y = Y_n$, we must have an LCS $Z_{k-1}$ of length $k-1$ for $X_{m-1}$ and $Y_{n-1}$, if $x_m = y_n$. So, our LCS $Z'$ should also be of length $k-1$. This resolves what we do when $x_m = y_n$, that is, we append an LCS of $X_{m-1}$ and $Y_{n-1}$ with $x_m$ to get an LCS of $X$ and $Y$.

The remaining difficulty lies in deciding what we must do if $x_m \neq y_n$. This case is also aided by the following Theorem 1. We closely follow the text [3].

## 1.1 Main LCS theorem

**Theorem 1.** *[3] Let $X = x_1, x_2, \cdots, x_m$ and $Y = y_1, y_2, \cdots, y_n$ be sequences, and let $Z = z_1, z_2, ..., z_k$ be any LCS of $X$ and $Y$.*

*1. If $x_m = y_n$, then $z_k = x_m = y_n$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$.*

*2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that $Z$ is an LCS of $X_{m-1}$ and $Y$.*

*3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that $Z$ is an LCS of $X$ and $Y_{n-1}$.*

**Proof**

(1) For the sake of contradiction, suppose $z_k \neq x_m$, then we could append $x_m = y_n$ to $Z$ to obtain a common subsequence of $X$ and $Y$ of length $k+1$, a contradiction, as $Z$ is already a longest common subsequence of $X$ and $Y$. So, $z_k = x_m = y_n$.

Furthermore, the prefix $Z_{k-1}$ is a length-$(k-1)$ common subsequence of $X_{m-1}$ and $Y_{n-1}$. We must to show that $Z_{k-1}$ is also an LCS indeed. Suppose for the purpose of contradiction that there is a common subsequence $W$ of $X_{m-1}$ and $Y_{n-1}$ with length greater than $k-1$. Then, appending $x_m = y_n$ to $W$ produces a common subsequence of $X$ and $Y$ whose length is greater than $k$, which is a contradiction.

(2) If $z_k \neq x_m$, then observe that $Z$ is a common subsequence of $X_{m-1}$ and $Y$. If there were a common subsequence $W$ of $X_{m-1}$ and $Y$ with length greater than $k$, then $W$ would also be a common subsequence of $X_m = X$ and $Y$, contradicting the assumption that $Z$ is an LCS of $X$ and $Y$.

(3) This case is similar but symmetric to the case (2) above.

□

## 1.2 The design of the algorithm from the main Theorem 1

If $x_m = y_n$, we must find an LCS of $X_{m-1}$ and $Y_{n-1}$. Appending $x_m = y_n$ to this LCS yields an LCS of X and Y. Otherwise, find an LCS of $X_{m-1}$ and $Y$, and find an LCS of $X$ and $Y_{n-1}$. The bigger one is the LCS of $X$ and $Y$. One optimal subproblem solution must be used within an LCS of $X$ and $Y$.

Let us define $c[i, j]$ to be the length of an LCS of the sequences $X_i$ and $Y_j$. If either $i = 0$ or $j = 0$, one of the sequences has length 0, so the LCS has length 0. Furthermore, the *optimal substructure* of the LCS problem renders the recursive formula $c[i, j]$=

(i) 0 if $i = 0$ or $j = 0$,

(ii) $c[i-1, j-1] + 1$ if $i, j > 0$ and $x_i = y_j$,

and (iii) $\max(c[i, j-1], c[i-1, j])$ if $i, j > 0$ and $x_i \neq y_j$.

## 1.3 The algorithm for computing the length of an LCS

Suppose we intend to only compute the length of the LCS. We can then use a matrix $c[m, n]$ which will store the computed values of the lengths $c[i, j]$'s of the LCSs of $X_i$'s and $Y_j$'s, for all

$1 \leq i \leq m$,$1 \leq j \leq n$.

$LCS-LENGTH(X,Y)$
$m = length[X]$ and $n = length[Y]$
for $i = 1$ to $m$ do $c[i,0] = 0$
for $j = 0$ to $n$ do $c[0,j] = 0$
for $i = 1$ to $m$ do for $j = 1$ to $n$
do if $x_i = y_j$
then $c[i,j] = c[i-1,j-1]+1$ and $b[i,j] = diag$
else if $c[i-1,j] >= c[i,j-1]$
then $c[i,j] = c[i-1,j]$ and $b[i,j] = up$
else $c[i,j] = c[i,j-1]$ and $b[i,j] = side$
return $c$ and $b$

### 1.4 Printing an LCS

In order the print an LCS, we can use the *diag*, *up* and *side* flags in matrix $b[m,n]$ as follows.

$PRINT-LCS(b,X,i,j)$
if $i = 0$ or $j = 0$ then return
if $b[i,j] = diag$ then $PRINT-LCS(b,X,i-1,j-1)$ and print $x_i$
elseif $b[i,j] = up$ then $PRINT-LCS(b,X,i-1,j)$
else $PRINT-LCS(b,X,i,j-1)$

## 2 The matrix chain problem

For $n$ matrices $A_i$, $1 \leq i \leq n$, let $p_{i-1}$ be the number of rows and $p_i$ be the number of columns of $A_i$. Consider the following (inefficient) recursive procedure that determines $m[i,j]$, the minimum number of scalar multiplications needed to compute the matrix-chain product $A[i..j] = A_iA_{i+1}\cdots A_j$. The following material is from [3].

### 2.1 Recursive inefficient code

$RECURSIVE-MATRIX-CHAIN(p,i,j)$
if $i = j$
then return 0
$m[i,j] = L$
for $k = i$ to $j-1$
do $q = RECURSIVE-MATRIX-CHAIN(p,i,k)$
$+RECURSIVE-MATRIX-CHAIN(p,k+1,j)+p_{i-1}p_kp_j$

if $q < m[i,j]$
then $m[i,j] = q$
return $m[i,j]$

## 2.2 The efficient polynomial time process

$MEMOIZED-MATRIX-CHAIN(p)$
1 $n = length[p] - 1$
2 for $i = 1$ to $n$
3 do for $j = 1$ to $n$
4 do $m[i,j] = L$
5 return $LOOKUP-CHAIN(p,1,n)$

$LOOKUP-CHAIN(p,i,j)$
1 if $m[i,j] < L$
2 then return $m[i,j]$
3 if $i = j$
4 then $m[i,j] = 0$
5 else for $k = i$ to $j-1$
6 do $q = LOOKUP-CHAIN(p,i,k) + LOOKUP-CHAIN(p,k+1,j) + p_{i-1}p_kp_j$
7 if $q < m[i,j]$
8 then $m[i,j] = q$
9 return $m[i,j]$

## 2.3 The efficient bottom-up iterative version

$MATRIX-CHAIN-ORDER(p)$
1 $n = length[p] - 1$
2 for $i = 1$ to $n$
3 do $m[i,i] = 0$
4 for $l = 2$ to $n$——— $l$ is the chain length.
5 do for $i = 1$ to $n-l+1$
6 do $j = i+l-1$
7 $m[i,j] = L$
8 for $k = 1$ to $j-1$
9 do $q = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$
10 if $q < m[i,j]$
11, 12 then $\{m[i,j] = q; s[i,j] = k\}$
13 return $m$ and $s$

### 2.4 Printing the parenthesized matrix chain

$PRINT-OPTIMAL-PARENS(s,i,j)$
1 if $i=j$
2 then print "$A''i$
3 else print "$(''$
4 $PRINT-OPTIMAL-PARENS(s,i,s[i,j])$
5 $PRINT-OPTIMAL-PARENS(s,s[i,j]+1,j)$
6 print "$)''$

## 3 Deterministic linear time selection by Blum et al., Prune-and-search technique [3]

We revise this method which was covered in the Algorithms I course in 2018.

### 3.1 The medians and median of medians scheme

We find the $i$th smallest of $n$ unsorted numbers in linear time. The median of medians $x$ is used to partition into the *smallers* set of $k$ items, and the *biggers* set of $n-k$ items.

### 3.2 The selection of the $i$ smallest by recursion after median of medians partition

We can see that there are at least $3(\lceil \frac{1}{2}\lceil \frac{n}{5}\rceil - 2\rceil)$ items larger than $x$, and also at least so many smaller than $x$. This lower bound is bigger than $\frac{3n}{10}-6$. So, the number of elements smaller than $x$ is no more than $n-(\frac{3n}{10}-6)=\frac{7n}{10}+6$. The second recursive call is over at most $\frac{7n}{10}+6$ elements and the first one is over $\lceil \frac{n}{5}\rceil$ elements.

### 3.3 Solving the recurrence

The recurrence is clearly
$T(n) \leq \Theta(1)$ if $n \leq 140$,
$T(\lceil n/5\rceil)+T(7n/10+6)+O(n)$ if $n>140$.

To show that the running time is linear, it is sufficient to show that $T(n)\leq cn$ for some suitably large constant $c$ and all $n>0$. Firstly, for a large enough $c$, $T(n)\leq cn$ holds for all $n\leq 140$. We also need another constant $a$, such that we can assign a specific function to the $O(n)$ term in the recurrence; this non-recursive component of the running time should be bounded above by $an$, for all $n>0$. Using this *inductive hypothesis* we get $T(n)\leq c(\lceil \frac{n}{5}\rceil)+c(\frac{7n}{10}+6)+an \leq (c\frac{n}{5}+c)+\frac{7cn}{10}+6c+an=\frac{9cn}{10}+7c+an=cn+(-cn/10+7c+an)$, which is at most $cn$ if $-cn/10+7c+an\leq 0$.

This inequality is $c\geq 10a(n/(n-70))$ when $n>70$. For $n>140$, $(n/(n-70))\geq 2$, and so choosing $c\geq 20a$ will satisfy our inequality for $n>140$.

## 4 Multiplying two $n$-bit numbers with $o(n^2)$ bit operations [1]

Divide-and-conquer is used so that recursive muliplcations are on numbers of half the size in terms of number of bits. So, if $x = \sqrt{N}a + b$ and $y = \sqrt{N}c + d$, where $a, b, c, d$ are $\frac{n}{2}$-bit numbers and $N = 2^n$, then we define $u = (a+b)(c+d)$, $v = ac$ and $w = bd$, so that $xy = Nv + \sqrt{N}(u - v - w) + w$. This requires only three $\frac{n}{2}$-bit multiplications in the recursion, thereby requiring atmost $3kn^{\log 3}$ bit operations solving the recurrence $T(n) = 3T(\frac{n}{2}) + kn$ for $n > 1$ and with basis case $T(1) = k$. The exact solution is $T(n) = 3kn^{\log 3} - 2kn$.

**Exercise 1:** What happens when $a+b$ or $c+d$ is a $\frac{n}{2}+1$ bit number due to carry in the additions?

## 5 Dynamic programming for computing a minimum length triangulation of a polygon

A convex polygon $P$ can be cut by a diagonal joning any pair of its $n$ vertices into two convex polygons. So, any such diagonal can be a triangulation edge. The number of possible triangulations is exponential. Finding a triangulation that minimizes the total length (or ink) for drawing the triangulation diagonals is our goal.

Let $P$ be a clockwise sequence (cycle) of vertices $v_1, v_2, \cdots, v_n$ For any triangulation the edge $v_n v_1$ is a base of some triangle $v_n v_1 v_k$, where $k \in \{2, 3, \cdots, n-1\}$. The problem $P(v_n, v_1)$ with triangle $v_n v_1 v_k$ fixed, has subproblems $P(v_n, v_k)$ and $P(v_k, v_1)$. The cost of these subproblems being $C(n,k)$ and $C(k,1)$, and the costs of $v_n v_k$ and $v_k v_1$ added, may be optimized over all $k \in \{2, 3, \cdots, v_{n-1}\}$.

Note that for $k = v_2$ and $k = v_{n-1}$, there is only one subproblem, the other one being vacuous. Also, one edge, respectively, $v_n v_2$ and $v_{n-1} v_1$ would add up.

## 6 Nearest subsequence

This discussion is about Tutorial 1. There is no match if $i > j$. So, $f(i,j) = 0$ if $i > j$. Also, $f(1,1) = |A(1) - B(1)|$. Moreover, $f(i,i) = f(i-1,i-1) + |A(i) - B(i)|$. We have more freedom in $f(i,j)$, where $j > i$; we can add $|A(i) - B(k)|$ over $f(i-1,k-1)$ for all $k$. $i \leq k \leq j$, and minimize over all these values of $k$. So, we take $f(i-1,i-1) + |A(i) - B(i)|$, $f(i-1,i) + |A(i) - B(i+1)|$, $f(i-1,i+1) + |A(i) - B(i+2)|, \cdots, f(i-1,j-1) + |A(i) - B(j)|$, and minimize over all these possibilities.

For $A = 2,4,7$ and $B = 3,2,6,9$, $f(0..3,0) = f(0,0..4) = 0$. Also, $f(1,1) = 1$, and $f(1,2) = f(1,3) = f(1,4) = 0$ because the single 2 in $A$ is aligned with the 2 in $B$ in all these cases. We know that $f(2,2) = 3$. Now, the non-trivial $f(2,3)$ and $f(2,4)$ cases may be considered. We know that $f(2,3) \leq f(2,2)$. However, we also need $f(1,1) + |A(2) - B(3)| = 1 + 2$ as well as $f(1,2) + |A(2) - B(3)| = 0 + 2 = 2$. So, $f(2,3) = 2$.

For $f(2,4)$ we try (i) $f(1,1) + |A(2) - B(2)| = 3, f(1,2) + |A(2) - B(3)| = 2, f(1,3) + |A(2) - B(4)| = 5$ and get $f(2,4) = 2$.

We know that $f(3,3) = 4$. For $f(3,4)$ we try $f(2,2) + |A(3) - B(3)| = 3 + 1 = 4, f(2,3) + |A(3) - B(4)| = 2 + 2 = 4$. So, $f(3,4) = 4$.

A smarter solution uses only $O(mn)$ time as uploaded in moodle CS31005 course page.

## 7 Divide-and-conquer methods for computing convex hulls of point sets in the plane

Here we consider two $O(n \log n)$-time algorithms for divide-and-conquer computation of convex hulls for a set of $n$ points in the 2d plane. The first one requires merging two vertically separated con-

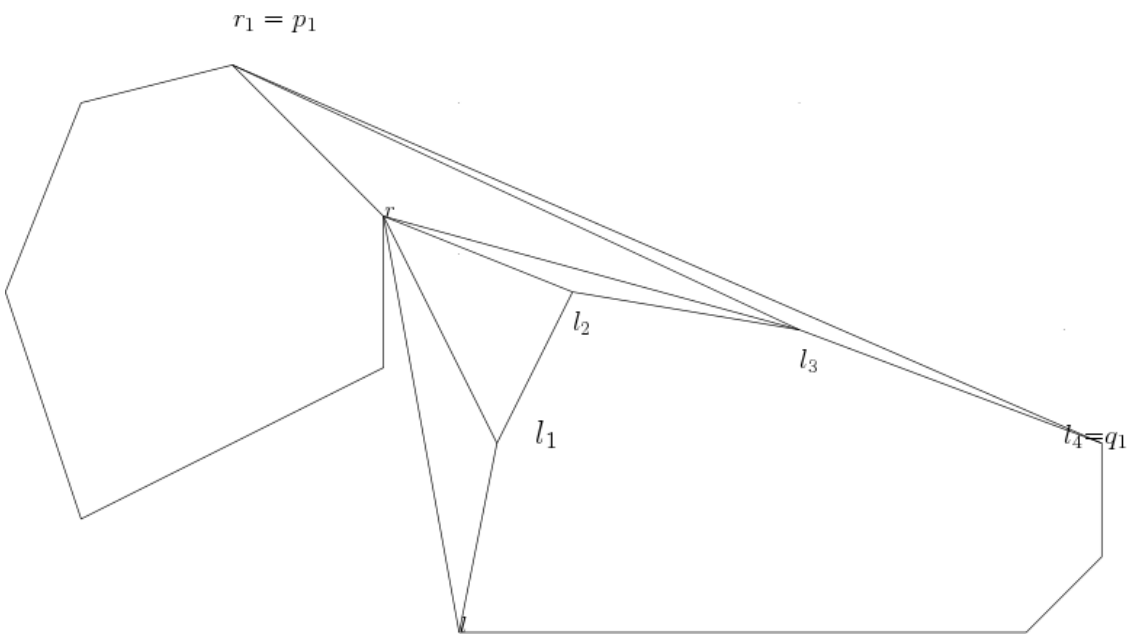


Figure 1: Computing the upper tangent $p_1q_1$

vex polygons in linear time by computing upper and lower common tangets. The second one merges two possibly intersecting convex polygons in linear time using the construction of a star-shaped polygon, whose conex hull is then computed using angular sweep by Graham's scan.

### 7.1 An algorithm that uses median finding or sorting to divide the given point set into two almost same sized sets [7]

Using median finding or by sorting (with x-coordinates of points), we first create the sets $S_1$ and $S_2$ where $S_1 \cap S_2 = \emptyset$ and $S = S_1 \cup S_2$ is the $n$ given points. Each of $S_1$ and $S_2$ has atmost $\lceil \frac{n}{2} \rceil$ points. For any set $S$ of points we say that $CH(S)$ is its *convex hull*. As a second step we compute the convex hulls $P_1 = CH(S_1)$ and $P_2 = CH(S_2)$. Note that these two convex hulls are vertically separated by the line $y = x_m$, where $(x_m, y_m)$ is a point in $S$ with the median from the set $\{x|(x,y) \in S\}$ being $x_m$.

In Figure 1, we start with the rightmost vertex $r$ of $P1 = CH(S1)$

and the leftmost vertex $l$ of $P2 = CH(S2)$. Finally we discover the upper common tangent $p_1q_1$; the lower common tangent is discovered using a symmetric and similar process. The sequence of events occuring in this transition of the segment from $rl$ to $p_1q_1$ is the discovery of the segments $rl$, $rl_1$, $rl_2$, $rl_3$, $r_1l_3$ and $r_1l_4 = p_1q_1$ that do not intersect the interiors of $P_1$ and $P_2$; indeed, these segments *traingulate* the region of the plane separating $P_1$ and $P_2$, which is bounded by the upper common tangent $p_1q_1$ and the starting segment $rl$.You will find this process in the algorithm explained in the text of O'Rourke [7]. Note that in each step, we reach a new vertex on $P_1$ or on $P_2$, which we reach exactly once. So, the running time complexity is linear.

Firstly, any vertex of $u$ of $P_1$ below $r$, when connected to any vertex $v$ of $P_1$ would cause $uv$ to "pierce" into the interior of $P_1$ and thus never be a tangent to $P_1$ at $u$. A symmetric argument eliminates any vertex $v$ of $P_2$ below $l$ from being a tangency point for $P_2$ for a segment $uv$ connecting any vertex $u$ of $P_1$ to $v$. So, our starting segment $rl$ is correct for beginning the search for an upper common tangent.

Observe by creating examples that one or both of the vertices of the upper common tangent may be different from the topmost vertices of $P_1$ and $P_2$.

Once the upper and lower common tangents are computed in linear time, we can work out the boundary representation of $CH(S_1 \cup S_2)$ in linear time.

Now consider the upper common tangent computation steps. The upper common tangent will connect only vertices on the upper parts of $P_1$ and $P_2$. This is established in Lemma 3.8.1 on page 93 of [7].

Let us call the segment connecting a vertex of $P_1$ and a vertex of $P_2$ as $T$, which is initially $rl$ and finally the upper common tangent $p_1q_1$. So, $T$ changes at one of its end or the other, at each step. However

$T$ never intersects the interior of $P_1$ nor of $P_2$. This is establiished in Lemma 3.8.2 in [7].

### 7.2 A linear time algorithm for computing the convex hull of all vertices of two given convex polygons

Finally, we use Graham's scan (for a star-shaped polygon) in this algorithm, where we first create the star-shaped polygon from the two given input convex polygons $P_1$ and $P_2$; the two input polygons may be overlapping in a their interior areas [9]. Graham's scan requires linear time. It requires an angular sorted list of points around a single point $p$. If we can get a point $p$ inside the intersection of the interiors of $P_1$ and $P_2$, we can then merge the two sorted angular lists for $P_1$ around $p$ and $P_2$ around $p$, thus getting a star-shaped polygon. If we get a point $p$ inside only one of $P_1$ and $P_2$, say inside $P_1$, then we drop a portion of $P_2$ and merge the rest of $P_2$ with $P_1$ to get an angular list, and thereby, a star-shaped polygon. See the text of Preparata and Shamos [9].

## 8 Closest pair of points in $\mathcal{R}^2$

We are given a set $S$ of $n$ points in the 2-d plane. We wish to compute the closest pair of points in $S$ in $O(n \log n)$ time. We base the following discussion on two texts [5, 9]. We compute $\delta_1$ and $\delta_2$, the closest pair distances inside sets $S_1$ and $S_2$, respectively. Here $S_1 \cup S_2 = S$ and $S_1 \cap S_2 = \emptyset$. The value $\delta$ is the smaller of $\delta_1$ and $\delta_2$. We must use $\delta$ in the conquer step to explore whether there is pair of points $(p,q)$, $p \in S_1$ and $q \in S_2$, such that the distance between $p$ and $q$ is less than $\delta$.

### 8.1 The main construction and its packing properties

Let $p \in S_1$ be any point in the left sub-rectangle of $R$, where $p$ lies equidistant from the top and bottom edges of the $2\delta \times 2\delta$ rectangle $R$. See Figure 2. Here, the rectangle $R$ is bisected by the vertical line

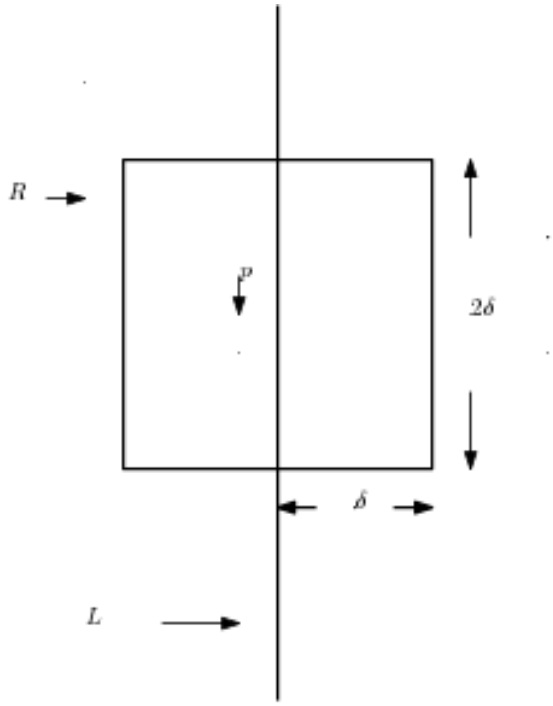


Figure 2: The point $p$ lying equidistant from the top and bottom edges of the rectangle $R$ can see its closest pair only inside the $\delta \times 2\delta$ sub-rectangle in the right half of the rectangle $R$.

that separates the two recursive subproblems defined over the point sets $S_1$ and $S_2$ of roughly $\frac{n}{2}$ points each, from $S = S_1 \cup S_2$, where $|S| = n$. Indeed, the closest point $q \in S_2$ for the point $p \in S_1$ can be only one of at most six points inside the right sub-rectangle of $R$ of size $\delta \times 2\delta$; observe that we cannot pack more than six points in this rectangle. Why? See [9].

### 8.2 The conquer process of finding the closest pair

We project all the points of $S^* \subseteq S$ on the dividing line $L$, where $S^*$ is the set of all points from $S$ that lie in the $2\delta$ corridor flanking the dividing line $L$. Now, we just need to argue that for each point $p$ in the $2\delta$ corridor, we need to check actual distances from above and below $p$, within the vertical projected distance of $\delta$ from $p$ above and below $p$, and this would discover at most 6 points for each point $p$.

See [9].

### 8.3 The overall algorithm and the divide step

We know that $S \setminus S^*$ has points outside the corridor of width $2\delta$. We can ignore these points and consider only the points in $S^*$ in the conquer step. Earlier we had computed $\delta_1$ and $\delta_2$, the closest pair distances inside sets $S_1$ and $S_2$, respectively. The value $\delta$, which is the smaller of $\delta_1$ and $\delta_2$, is what we must use in the conquer step to explore whether there is pair of points $(p,q)$, $p \in S_1$ and $q \in S_2$, such that the distance between $p$ and $q$ is less than $\delta$.

### 8.4 Implementation and the time complexity

Consider the sorted list of $S^*$, sorted by the $y$-coordinates of the projections of the points of $S^*$ on the dividing line $L$. We process each $p \in S^* \setminus S_2$ in the sorted order looking for all $q \in S^* \setminus S_1$, with vertical separation at most $\delta$ from $p$ along $L$, both above and below $p$. There can be at most 6 such points $q$. The whole process in the conquer step is therefore linear.

### 8.5 An alternative argument as per [5]

Assuming that $s$ and $s'$ are two points that have separation less than $\delta$, we wish to show that given $s$, $s'$ can be picked up by doing only a few steps of computations. Here, we have the square of $2\delta$-sized edges symmetrically flanking the dividing line $L$. This square can be viewed as comprising 16 squares of sides of length $\frac{\delta}{2}$, 8 on each side of $L$. Each such small square can have at most one point. Why? So at most 16 points can be there in the big square of sides $2\delta$. If $s$ and $s'$ are inside this big square and their separation is less than $\delta$ then $s$

being above $s'$, we will need to check at most 15 points below $s$ in vertical sorted order of all points in the $2\delta$ corridor. If we need to see 16 or more points then we spend at least 3 rows or more than 12 points and thus introduce a separation of $\frac{3}{2}\delta$, a contradiction.

## 9 Binary space partioning (BSP) trees

We may study this topic towards the end of this course. We base our discusssions in this section on two texts [2, 6]. When we build the *BSP* tree, we just wish to bound the number of *parts* of segments *cut* by previously introduced segments; segments are entered one by one from a set $N$ of $n$ segments. The number of *fragments* created from segments appearing later in the insertion order is the number of nodes of the created BSP tree. Therefore, the order in which these segments arrive is important, which we consider to be a random permutation of the $n$ segments. In this scenario, we wish to determine the expected number of fragments of the $n$ segments. If not ordered carefully, we may get $O(n^2)$ fragments in the worst case. Why? So, estimating the expected number of fragments is our objective.

Each cut results in just one new fragment. So, it is sufficient to count the number of cuts. Any newly introduced segment $S \in N$ can potentially cut each segment introduced after $S$. We wish to determine the probability that $S$ cuts $S'$, where the pair $(S, S')$ is any fixed ordered pair of segments from $N$.

Now consider the relative locations of the $n(S, S')$ segments in $N$ that come on the line $l(S)$ between $S$ and $S'$. All these $n(S, S')$ segments as well as the segment $S'$ must be introduced after $S$ is introduced, so that $S$ can cut $S'$, as well as all these $n(S, S')$ segments. So, it does not matter in which relative order the $n(S, S')$ intermediate segments and $S'$ are introduced as long as $S$ comes before all these

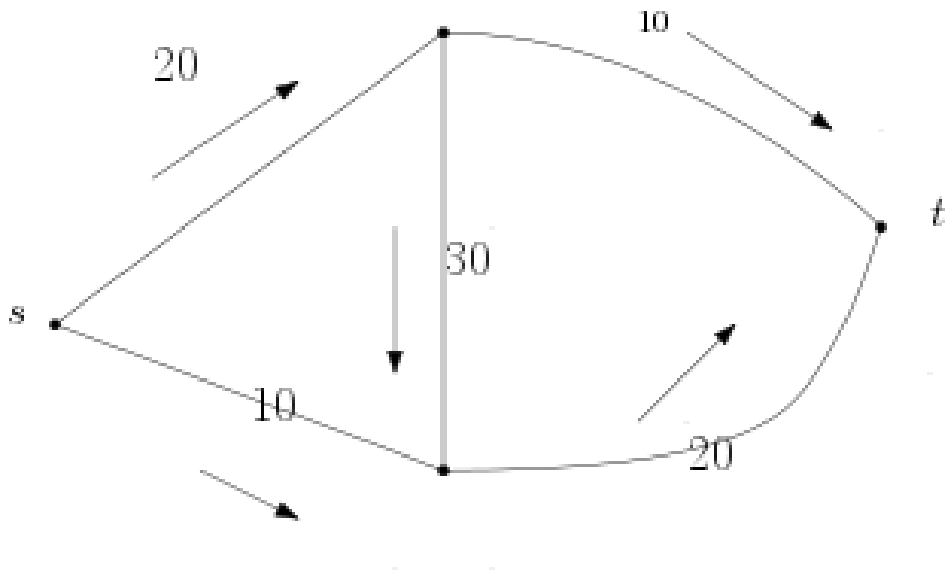


Figure 3: The network of four vertices and five directed edges with capacities.

$n(S,S')+1$ segments in arrival order. The *conditional* probability with which this happens is clearly $\frac{1}{n(S,S')+2}$, given that $S$ and $S'$ have been fixed apriori for such an event.

The final step is now to determine the expected number cuts. The sum of expectations rule can be applied to the events where $S$ cuts $S'$, over all ordered pairs $(S,S')$ . So, as in [6], the sum of expectations of these individual events is $\sum_{(S,S')} \frac{1}{n(S,S')+2}$. We will keep the clearance for some other segments appearing before $S$ so that instead of an equality, this sum, which can be easily seen to be $O(n\log n)$, may be used as an upper bound on the final expectation for the total number of cuts.

## 10 Network flow computations

We develop the basics for network flow computations, as in [3] from its section 26.2. We begin with a four node network with source $s$ and sink $t$. An initial flow of 20 units is indicated for the network of Figure 3 in Figure 4. Later the flow

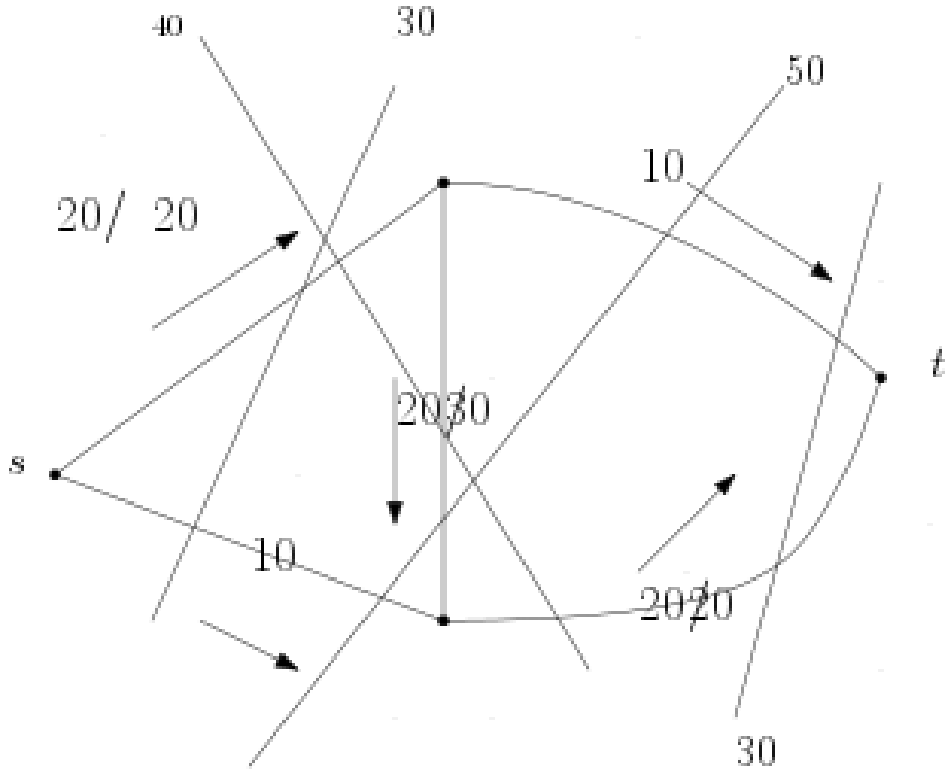


Figure 4: The cuts $(S,T)$ where $S$ has the source and $T$ has the sink, are shown as labelled lines with cut capacities.

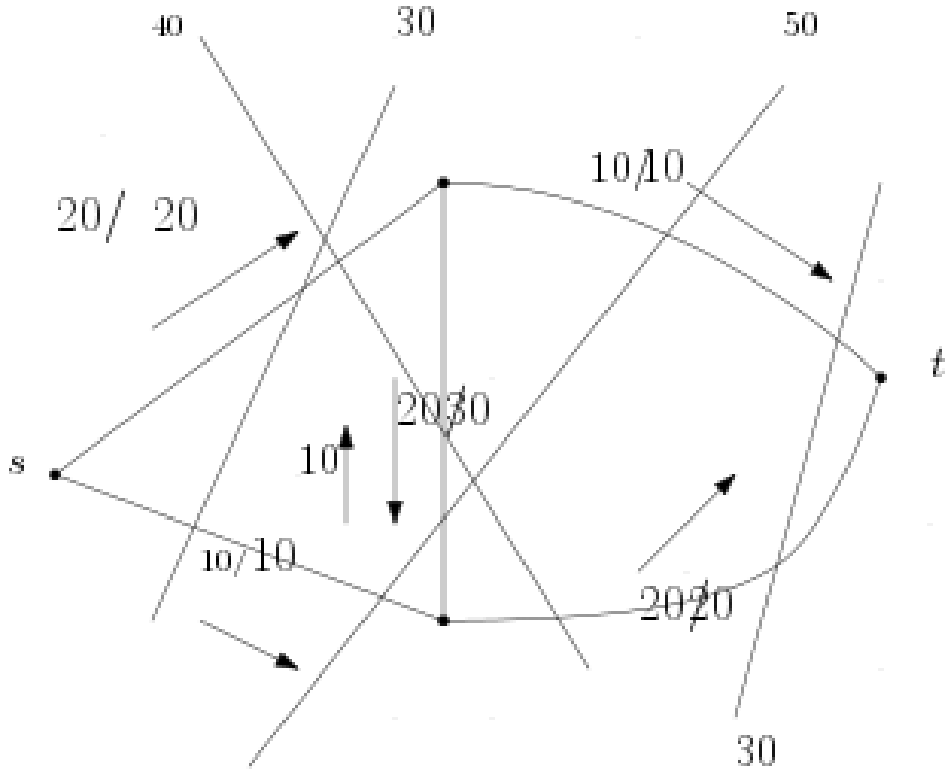


Figure 5: Flow of 10 units is reversed to reduce net flow to 10 from 20 in the middle link so that the total flow reaches the maximum possible limit of 30 units, as indicated by the minimum capacity cuts.

across the middle link is reversed by 10 units to reduce the total flow in that link downwards from 20 to 10 units. This helps accomodate an additional flow of 10 units, matching the capacity of the two (minimum) cuts of capacity 30 units. This makes it possible to achieve the maximum flow assignment of 30 units as in Figure 5 for this network.

Study the generic flow augmentation step on residual networks, and the Ford-Fulkerson method, its analysis and drawbacks as in section 26.2 of [3]. For the Edmonds-Karp method of augmenting by shortest paths in the residual networks, we show (based on the treatment done as in [3]) that the shortest distance $\delta_f(s,v)$ of $v$ from the source $s$ in $G_f$ never falls for subsequent flows (see Lemma 1). Using this property, we show that between two "appearances" of an edge $(u,v)$ in the residual networks $G_f$ and $G_{f'}$, for flows functions $f$ and $f'$, respectively, the shortest path from $s$ to $u$ increases by at least two units, that is, $\delta_{f'}(s,u) \geq \delta_f(s,u)+2$. This is easy to see; some edge $(u,v)$ becomes "critical" in a residual network $G_f$ due to an augmentation along an augmenting path in $G_f$ where $(u,v)$ is the "bottleneck" or "critical" edge, and consequently $(u,v)$ vanishes from the subsequent residual network. The resulting flow after this augmentation is of course $|f|+c_f(u,v)$. So, we have

$$\delta_f(s,u) = \delta_f(s,v) - 1 \tag{1}$$

along the shortest path along which the augmentation was done in $G_f$. Further, this same vanished edge $(u,v)$, which vanished after $G_f$ in $G_{|f|+c_f(u,v)}$, can "reappear" in a subsequent residual network, only when the flow along the directed edge $(v,u)$ will be increased from zero to a finite value in some residual network, after some future flow function $f'$ is already achieved. That is, in the residual network $G_{f'}$ of $f'$, this "reverse" flow will cause $(u,v)$ to "reappear" in the very subsequent residual network $G_{|f'|+c_{f'}(v,u)}$, subsequent to $G_{f'}$. During the reverse flow in this augmentation in $G_{f'}$, we have

$$\delta_{f'}(s,v) = \delta_{f'}(s,u) - 1, \tag{2}$$

So, using inequalities 1 and 2, we have an increase of at least two units from $\delta_f(s,u)$ to $\delta_{f'}(s,u)$ because we assume that $\delta_{f'}(s,v)$ is not smaller than $\delta_f(s,v)$ as stated in Lemma 1 in the inequality 3.

**Lemma 1** (Monotonicity lemma)**.** *Let $f$ be a valid flow function for a network $G(V,E)$ with source $s$ just after augmentation at a certain stage in the Edmonds-Karp method. Let $f'$ be the valid flow function for $G(V,E)$ in a subsequent stage*

*of augmentation. Then, for any vertex $v \in V$, we have*

$$\delta_f(s,v) \leq \delta_{f'}(s,v), \tag{3}$$

*that is, the shortest path distance from $s$ to $v$ in $G_f$ is no more than that in $G_{f'}$.*

**Proof**

It is enough to consider two consecutive flow functions $f$ and $f'$ for $G(V,E)$ in the Edmonds-Karp augmentations. For the sake of contradiction, let $v$ be the closest vertex to $s$ in $G_{f'}$ such that $\delta_f(s,v) > \delta_{f'}(s,v)$, and let $u$ be the previous vertex of $v$ in the shortest path from $s$ to $v$ in $G_{f'}$. So, $\delta_{f'}(s,u) = \delta_{f'}(s,v) - 1$. Also, $u$ being closer to $s$ than $v$ in $G_{f'}$, we have $\delta_f(s,u) \leq \delta_{f'}(s,u)$. Eventhough $(u,v) \in E_{f'}$, $(u,v)$ may or may not be an edge of $G_f$. Note that if $(u,v) \in E_f$ in $G_f$, the shortest distances to $u$ and $v$ would differ by at most 1, and thus we could have written $\delta_f(s,v) \leq \delta_f(s,u) + 1 \leq \delta_{f'}(s,u) + 1 = \delta_{f'}(s,v)$, a contradiction to the assumption that $\delta_f(s,v) > \delta_{f'}(s,v)$. So, we assume that $(u,v)$ is not in $E_f$. However, $(u,v)$ reappears in $G_{f'}$. So, there must have been a back flow in the augmentation in $G_f$, resulting in flow $f'$, from $v$ to $u$ so that $(u,v)$ reappears in $G_{f'}$, after having been absent in $G_f$. So, in this augmentation in $G_f$, we have $\delta_f(s,v) = \delta_f(s,u) - 1 \leq \delta_{f'}(s,u) - 1 = \delta_{f'}(s,v) - 2 < \delta_{f'}(s,v)$, a contradiction again.

□

# 11 The theory of NP-competeness and NP-complete problems

Firstly, we should try to establish that a certain problem is in the class *NP*, if we are unable to show that it is in the class *P*. The class *NP* is defined for polynomial time or depth, whereas computation is not deterministic for *NP*, unlike the case of deterministic polynomial time or depth computation in the case of the class *P*. The essential additional non-deterministic feature makes all the difference. So, we have a plethora of decision problems that are not known to be decidable in *P* but are in the class *NP*. Problems such as SAT, 3SAT (3CNF-SAT), CIRCUIT-SAT are such problems. Whereas the CIRCUIT-VALUE problem is in *P*, its non-deterministic counterpart CIRCUIT-SAT is shown to be in *NP* as well as *"complete"* for the class *NP*. Not surprisingly, CIRCUIT-VALUE is "complete" for the class *P*, just as CONTEXT-FREE-EMPTINESS is "complete" for the class *P*. Being "complete" for the class *NP* is known as NP-completeness. For definitions and discussions about these classes, see the standard texts [3, 4, 1, 8]

## 11.1 NP-completeness of the vertex cover and independent set problems

The *vertex cover* problem is that of selecting a small subset of vertices of the graph that *hit* or *cover* all edges. Naturally, we are happy to ask the following question: do $k \leq n$ vertices form a vertex

cover for the given $n$-vertex graph? Here, the inputs are $k$ and the graph $G(V,E)$ itself. We can also ask whether $G(V,E)$ has an *independent set* of size at least $k$, given $G(V,E)$ and $k$ as inputs. So, the vertex cover and the independent set problems related, and the NP-completeness of one implies that of the other. You may also see pages 460-462 of [5] for an NP-completeness proof for the independent set problem.

The vertex cover problem is also very useful in showing other path related problems NP-complete, as we see in Sections 11.3 and 11.4. A vertex cover is a set of vertices that give access to edges, and edges make paths.

### 11.1.1 The precise decision version statement and the construction

The NP-hardness reduction from 3-SAT to the vertex cover problem (and also to the independent set problem) may proceed by constructing a graph $G_f(V_f,E_f)$ for any given 3-SAT CNF formula $f$, such that $f$ is satisfiable if and only if $G_f$ has a vertex cover of size exactly $k = 2|V_f|/3$. Here $V_f$ is the set of vertices, one for each of the $3m$ literals from the $m$ clauses in $f$. The set $E_f$ of edges form triangles for each clause; three edges between pairs of literals in each clause. In this very same construction, we also observe that $f$ is satisfiable if and only if $G_f(V_f,E_f)$ has an independent set of size $m$. See Figure 6. More edges join inconsistent pairs of literals across the clause triangles, like $x_i$ with $x_i'$. Note that the minimum vertex cover must have size at least $2/3|V_f|$.

### 11.1.2 The only-if-part

Suppose $f$ is a satisfiable formula. We show that every minimum vertex cover of $G_f$ will have exactly $2/3|V_f| = 2m$ vertices. In other words, we show that $2m$ vertices are both necessary and sufficient for making the minimum cardinality vertex cover; we need at least $2m$ vertices to cover all the $3m$ edges of $m$ triangles. Let us consider any truth assignment $U$ for $f$. The function $U$ assigns a value 'T' or 'F' to each variable of $f$, thereby assigning true or false value to each literal in each clause. In $G_f$, we have one vertex for each literal of each clause, totalling to $3m$ vertices in all.

We know that the satisfying truth assignment $U$ falsifies at most two literals in each clause, leaving at least one literal in each clause satisfied. Consider the cross edges in $G_f$ that run between the triangles. We must choose two vertices from each triangle so that all these cross edges are also covered; two vertices per triangle will in any case cover the edges of the triangles. Since each cross edge can be covered from either end, we choose the end vertex whose corresponding literal is falsified by the truth assignment $U$; exactly one vertex of such a cross edge will correspond to a falsified literal. We thus cover all cross edges by choosing at most two vertices per triangle because each clause corresponding to any triangle has at most two falsified literals. If we have selected no vertex (or only a single vertex) from some triangle then we simply select two (or one) additional vertex from the traingle arbitrarily, so that a total of two vertices are selected from each triangle. This shows the existence of a vertex cover of minimal cardinality $2m$ in $G_f$.

In Figure 7, let $U$ denote 'T' for X1 and X4, and 'F' for X2 and X3. In the left clause we select falsified nodes X1' and X3 as vertex cover vertices. In the right clause, we select falsified nodes X2 and X4', and in the top clause X3, as the falsified clauses. However, we choose one more node

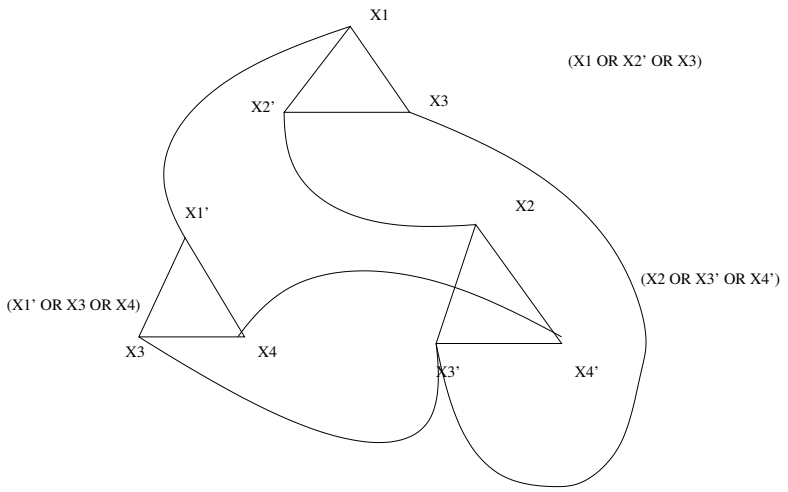


Figure 6: The formula and the graph

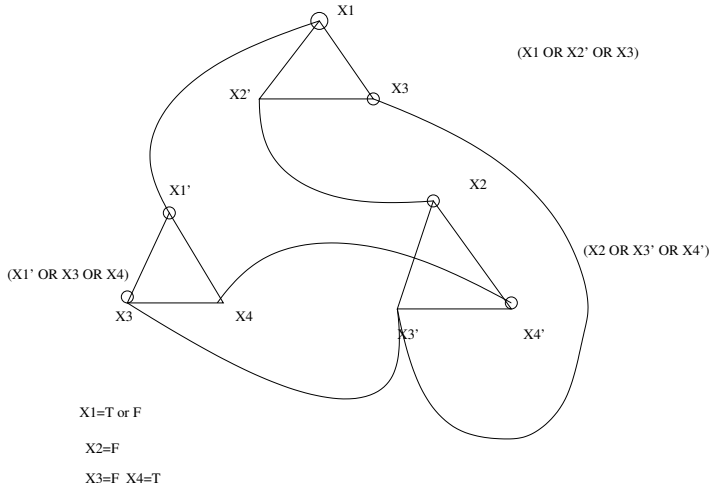


Figure 7: From a truth assignment to a vertex cover

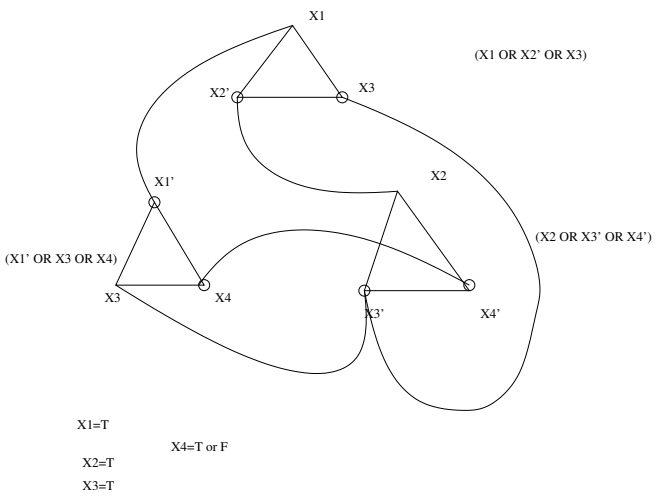


Figure 8: From a vertex cover to a truth assignment

in the top clause, even if it is not falsified, that is, node X1. So, the vertex cover marks one edge connecting the top and left clauses at both ends and all other cross edges from just one end.

### 11.1.3 The if-part

Conversely, suppose $G_f$ has a minimum vertex cover $C$ of size exactly $2/3|V_f| = 2m$. We show that $f$ must be satisfiable; we generate a truth assignment $U$ that satisfies at least one literal in each clause. The vertex cover $C$ has exactly two vertices in each triangle and certainly covers each cross edge. So, any cross edge $e$ is covered at least at one of its ends by some vertex in $C$. We inspect the triangles one by one; for each triangle $S$ we consider the vertex $v$ that is not in the vertex cover and assign a truth value 'T' to the variable at that vertex $v$ if the literal $l(v)$ at that vertex is uncomplemented, and 'F', otherwise. This will make at least one literal satisfied in the clause of the triangle $S$, but falsify literal(s) in other clauses if the vertex $v$ is connected by cross edges to other clause triangles. So, such actions for all triangles can falsify at most two literals per clause. Thus, a truth assignment is constructed for the formula $f$, since the single vertices selected from the triangles are not in the given vertex cover of size $2m$ (the selected vertices form an independent set with no cross edges between them!), thereby realizing a consistent (satisfying) truth assignment for the formula $f$.

Exercise: Determine a truth assignment for the example in Figure 8.

Hint: Pick X1 from the top clause, X3 from the left clause and X2 from the right clause in Figure 8. For the different vertex cover as given in Figure 7, select nodes for X2', X4 and X3' from the top, left and right clauses.

## 11.2 NP-completeness of the clique problem

The complement set $V \setminus C$ of a *vertex cover* $C$ of a graph $G(V,E)$ is an *independent* set of $G(V,E)$. The complement set $V \setminus C$ is a *clique* in the complement graph $\overline{G}(V,\overline{E})$. So, we can say that $G(V,E)$ has a vertex cover of size at most $k$ if and only if $\overline{G}(V,\overline{E})$ has a clique of size at least $|V| - k$. Thus

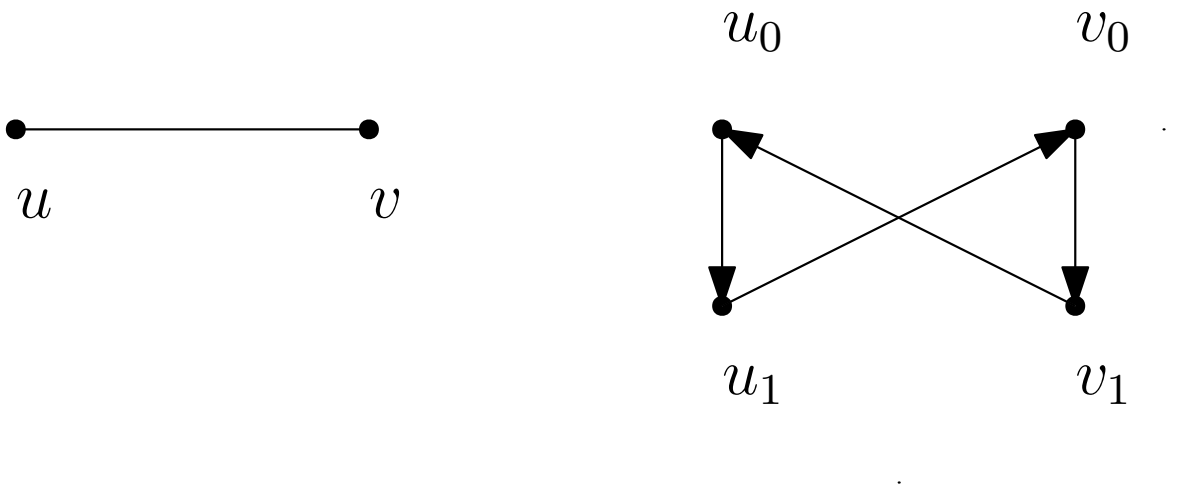


Figure 9: Edge $\{u, v\}$ in $G$ and four directed edges in $D$ for the construction in Section 11.3

the NP-hardness of the vertex cover problem implies NP-hardness of the clique problem, and vice versa.

We allude to the reduction from the 3-SAT problem to the clique problem as given in [3]. This simple reduction involves the construction of a graph $G(V_f, E_f)$ for any given formula of $m$ clauses in $n$ variables, such that the graph $G_f$ has a clique of size $m$ if and only if $f$ is satisfiable. The clauses need not be having exactly three literals; some may have even more literals. The $m$-partite graph $G_f$ has as many partites as clauses and no edges between vertices that correspond to literals in the same clause; one vertex is mapped to each literal of each clause. An edge is drawn between vertices corresponding to literals in two different clauses provided the two literals are not complements of each other. It is now easy to see that there is a clique of size $m$ in the graph if and only if the formula is satisfiable.

## 11.3 NP-completeness of the feedback edge set problem

Let us consider all directed cycles in a directed graph $D$, and try to find a small set of (directed) edges that contains at least one (directed) edge from each (directed) cycle of the $D$. Such as set is called a *feedback edge* set for $D$. We wish to address the question of determining whether a given directed graph $D$ has a feedback edge set of size at most a given integer $k$. We use a polynomial time reduction from the decision version of the vertex cover problem.

### 11.3.1 The construction

We follow the presentation in [1]. Given an undirected graph $G(V, E)$ and an integer $k$, we show that it is possible to construct a directed graph $D(V \times \{0, 1\}, E')$ in polynomial time in the size of $G$ and $k$, such that $D$ has a feedback vertex set of size $k$ if and only if $G$ has a vertex cover of size $k$. There are many nice ideas in this construction that use the edge covering ideas for vertex covers in undirected graphs and the cycle hitting properties of directed edges in directed graphs.

We need small feedback edge sets. So, we construct $D$ from $G$ in such a way that cycles

in $D$ will be forced to pass through certain edges in $D$ that are localized between vertices in $D$ corresponding to the same vertex in $G$. Also, cycles in $D$ will be based on edges in $G$. So, we will construct two vertices in $D$ for each vertex in $G$ and four edges in $D$ for each edge in $G$. For any vertex $v$ in $G$ we will have an *upper* and a *lower vertex* $v0$ and $v1$, resopectively, in $D$. For each edge $\{u,v\} \in G$, we will have *rising* edges $(v1,u0),(u1,v0)$, and falling edges $(u0,u1)$ and $(v0,v1)$ (see Figure 9). Note that symmetry makes $\{u,v\}$ and $\{v,u\}$ represented by the same set of 4 vertices and 4 edges in $D$. Naturally, the alternating rising and falling edges form a cycle in $D$ for each edge in $G$. So, a feedback vertex set $F$ for $D$ must have at least one edge from each of these 4-edge cycles of $D$ as well. Furthermore, all cycles in $D$ are of even length.

### 11.3.2 The characterization

We note that the number of edges in $D$ is $|E'| = 4|E|$. In $D$, let us replace each rising edge $(v1,w0)$ in the feedback edge set $F$ by the falling edge $(w0,w1)$. This may reduce the number of edges in the resulting the set $F' \subseteq F$ but $F'$ remains a feedback edge set for $D$. Now we show that $D$ has a feedback edge set of $k = |F'|$ edges if and only if $G$ has a vertex cover of size $k$.

Here, $F'$ is the feedback edge set of size $k$ as mentioned above. So, all cycles in $D$, including the 4-cycle corresponding to any edge $\{u,v\}$ of $G$, must be having a falling edge $(u0,u1) \in F'$ or $(v0,v1) \in F'$ in $D$, and we may include $u$ or $v$, respectively in the vertex cover for covering edge $\{u,v\}$ of $G$. So, $G$ has a vertex cover of size $k$. This completes the (only-if-part) of the characterization.

(if-part) Let $S$ be a vertex cover of size $k$ in $G$. Each edge of $G$ is covered from at least one of its ends by a vertex in $S$. So, we may write an edge of $G$ as $\{v,w\}$, where $w \in S$. For each $w \in S$, take $(w0,w1)$ in $D$ as a set $F'$ of $k$ edges in $D$. We wish to show that $F'$ is a feedback edge set for $D$. Observe that any rising edge $(v1,w0)$ in $D$ (where $w \in S$), of any cycle $C$ in $D$ is replaced by the falling edge $(w0,w1)$ in that same cycle $C$ as this is the only edge through which the cycle $C$ can move from $w0$ in $D$. Also, any rising edge $(w1,v0)$ (where $w \in S$), of any cycle $C$ in $D$, must be preceded by a falling edge $(w0,w1)$ in $D$ in the same cycle $C$ as the only incoming edge to $w1$ in $D$ is from $w0$. So, $F'$ is a feedback edge set for $D$ of size $k$.

## 11.4 The NP-completeness of DHC, the directed Hamiltonian circuit problem

As in [1], given an undirected graph $G(V,E)$ and an integer $k$, we discuss the construction (i) of a directed Hamiltonian circuit in a directed graph $D$ from a vertex cover of size $k$ in the given graph $G$, and (ii) of a vertex cover in $G$ of size $k$ from any directed Hamiltonian Circuit (abbreviated as HC henceforth) in $D$. Here, $D$ is designed based on $G$ and $k$ in polynomial time.

### 11.4.1 Mapping vertex covers to tours and vice versa

The construction of the directed graph $D$ involves making as many lists of edges as we have vertices in $G$; each list has a length proportional to the degree of the corresponding vertex therefore. An HC in $D$ will have to pass through all these vertex lists. Edges $\{u,v\}$ of $G$ are realized in $D$ using (i) two *upper and lower* vertices $u0(v)$ and $u1(v)$ for $u$, and two *upper and lower* vertices $v0(u)$

and $v1(u)$ for $v$, and
(ii) six directed edges in $D$ viz., *falling edges* $(u0(v), u1(v))$ and $(v0(u), v1(u))$, *forward cross edges* $(u0(v), v0(u))$ and $(u1(v), v1(u))$, and *backward cross edges* $(v0(u), u0(v))$ and $(v1(u), u1(v))$. An HC in $D$ will have to pass through all the four vertices for each edge in $G$.

We need $k$ vertices in $D$, where these $k$ *special* vertices are named $a_1, a_2, ..., a_k$. Recall that $k$ is the input along with the undirected graph $G(V,E)$ for the vertex cover problem.

Each $a_i$ in $D$ has an edge directed to the starting of each of the $|V|$ vertex lists. The last vertex of each vertex list has a directed edge to each $a_i$. An HC in $D$ needs to pass through each $a_i$ entering it only once and leaving it only once. So, an HC starting at $a_1$ will go through the starting point of just one vertex list and also leave the same vertex list to go to another $a_i$ and then repeat the same process till all vertex lists are covered and all $a_i$ are covered returning to $a_1$.

Consider the vertex list for each vertex $u$ of $G$. If $v$ and $w$ are adjacent to $u$ in $G$ and also consecutively appearing in the vertex list for $u$ in $D$ then there is a directed *falling* edge in $D$ from $u1(v)$ to $u0(w)$. These falling edges, the above earlier defined falling edges, and the forward and backward cross edges defined earlier, along with the edges incoming and out going to all $a_i$, $1 \leq i \leq |V|$, constitute all the edges of $D$.

### 11.4.2 The characterization

We show that $D$ has an HC if and only if $G$ has a vertex cover of size $k$. If there is a vertex cover $S \subseteq V$ of size $k$ in $G$ then we can initially make a cycle passing through only the $k$ corresponding vertex lists and the $k$ vertices $a_i$, $1 \leq i \leq k$. This cycle may initially pass through only vertices like $u0(v)$ and $u1(v)$ passing through edges like $(u0(v), u1(v))$, where $v$ is adjacent to $u$ in $G$. If additionally, $v$ is covered by $u$ by virtue of $u$ being in the vertex cover $S$ of $G$ then we can replace $(u0(v), u1(v))$ by the detour $(u0(v), v0(u))$, $(v0(u), v1(u))$, $v1(u), u1(v)$. Observe that in doing such detours, all vertices of $D$ would be tied up in a HC. This completes the part (i) about constructing an HC given a vertex cover of size $k$ for $G$, the if-part of the characterization.

Now consider part (ii) where we have an HC of $D$ and we wish to construct a vertex cover for $G$ of size $k$. Cearly, there will be $k$ paths separated by the $k$ special $a_i$ vertices in any HC for $D$. Each such path will give us one vertex $u$ of $G$ whose list would be traversed with possible detours for one or more vertices $v$ covered by $u$ in $G$. We collect such $k$ vertices like $u$ and observe that they constitute a vertex cover in $G$. This completes the part (ii), the only-if-part of the characterization.

## 11.5 The NP-completeness of the undirected Hamiltinian circuit problem UDHC

As in [1], we demonstrate a reduction from HDC to UDHC as follows.

## 11.6 The Cook-Levin result

As in [3], and as presented in class, CIRCUIT-SAT is first shown to be NP-complete and then this problem is reduced in steps to 3SAT.

# References


[1] A. V. Aho, J. E. Hopcroft, J. D. Ullman, The design and analysis of computer algorithms, Second impression, 2007.

[2] Mark de Berg, Otfried Cheong, Marc van Kreveld and Mark Overmars, Computational Geometry: Algorithms and Applications, Third Edition, Springer, 2008. (Earlier editions 1997, 2008, 2000).

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, Introduction to algorithms, Second Edition, Prentice-Hall India, 2003.

[4] J. Hopcroft and J. D. Ullman, Introduction to Automata, Languages and Computation, Addison-Wesley, 1979.

[5] J. Kleinberg and E. Tardos, Algorithm design, Pearson education, 2006.

[6] Ketan Mulmuley, Computational Geometry: An Introduction Through Randomized Algorithms, Prentie-Hall, 1994.

[7] J. O'Rourke, Computational Geometry in C, Second Edition, Cambridge University Press, 1997.

[8] C. H. Papadimitriou, Computational Complexity, Addison-Wesley Publishing Company, 1994.

[9] F. P. Preparata, M. I. Shamos, Computational Geometry: An Introduction, Springer, 1985.