Algorithms II

Selected Lecture Notes

Partha Bhowmick

IIT Kharagpur

http://cse.iitkgp.ac.in/~pb

Autumn 2013

Contents

Chapter 1

Divide and Conquer

Believe it can be done. When you believe something can be done, really believe, your mind will find the ways to do it. Believing a solution paves the way to solution.

— **David Joseph Schwartz**

A *divide-and-conquer* algorithm recursively breaks down a problem into two or more sub-problems of the same type, until they become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

Example: Binary search, Quicksort, Merge sort, Multiplication of two complex numbers, Integer multiplication, Matrix multiplication (Strassen's algorithm), Closest pair in a point set, Convex hull, etc.

1.1 Integer Multiplication

Consider the computation involved in finding $(a + bi)(c + di)$. If we express it as $(ac - bd) + (bc + ad)i$, then we need 4 scalar multiplications. But, as $bc + ad = (a + b)(c + d) - ac - bd$, so it can be done with just three: $ac, bd, (a + b)(c + d)$. Historically, it was first noticed by the mathematician, Carl Friedrich Gauss (1777–1855).

We can extend the above idea to multiply two binary numbers a and b. Let, for simplicity, a and b have n bits each, and n is a power of 2. Then we can express a as a concatenation of two binary numbers, a_L followed by a_R, each of length $n/2$ bits, so that a "looks like" $a_L a_R$ and its value is $a = 2^{n/2}a_L + a_R$. Similarly, $b = 2^{n/2}b_L + b_R$. Thus, $ab = (2^{n/2}a_L + a_R)(2^{n/2}b_L + b_R) = 2^n a_L b_L + 2^{n/2}(a_L b_R + a_R b_L) + a_R b_R$, which can be computed from the four $n/2$-bit numbers (a_L etc.) by three multiplications between $n/2$-bit numbers: two for $a_L b_L$ and $a_R b_R$, and one for $a_L b_R + a_R b_L$, since $a_L b_R + a_R b_L = (a_L + a_R)(b_L + b_R) - a_L b_L - a_R b_R$.

1.1.1 Time complexity

Addition of two numbers takes linear time. Multiplication by a power of 2 also takes linear time. Hence,

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n) = O(n^{\log_2 3}) \approx O(n^{1.58}).$$

1.2 Closest Pair

Given a set of points $P = \{p_1, \ldots, p_n\}$, the problem is to find the minimum distance $\delta_{\min}$ between two points of P. The naive algorithm is to consider all $\binom{n}{2}$ point-pairs of P, compute their distances, and find the minimum of these $\binom{n}{2} = O(n^2)$ distances, which needs $O(n^2)$ time. An efficient algorithm based on divide-and-conquer approach[1] is given below, which needs $O(n \log n)$ time.

The divide-and-conquer algorithm is as follows.

1. Sort the points of P to P_x and to P_y using x- and y-coordinates, respectively. (This step is out of recursion.)

2. Partition P into P_L and P_R by the vertical median-line $l_\mu : x = x_\mu$, using P_x.

3. Recursively compute the minimum distances δ_L and δ_R for P_L and P_R, respectively.

4. $\delta' \leftarrow \delta \leftarrow \min(\delta_L, \delta_R)$.

5. Traverse P_y and *append* a point $(x, y) \in P_y$ to Q_y (initialized as empty) if $x \in [x_\mu - \delta, x_\mu + \delta]$. If $x \in [x_\mu - \delta, x_\mu]$, then mark it as GRAY; otherwise BLACK (Fig. 1.1b).

   Result: $Q_y (= Q_L \cup Q_R)$ is y-sorted.

6. Traverse Q_y in order from first to last, and for each GRAY point $p_L \in Q_y$,

   (a) compute the distances of *four* BLACK points following p_L and *four* BLACK points preceding p_L in Q_y;

   (b) find the minimum δ'' of the above eight distances;

   (c) $\delta' \leftarrow \min(\delta', \delta'')$.

7. *return* $\min(\delta, \delta')$.

[1] M.I. Shamos and D. Hoey. Closest-point problems. *Proc. FOCS*, pp. 151–162, 1975.

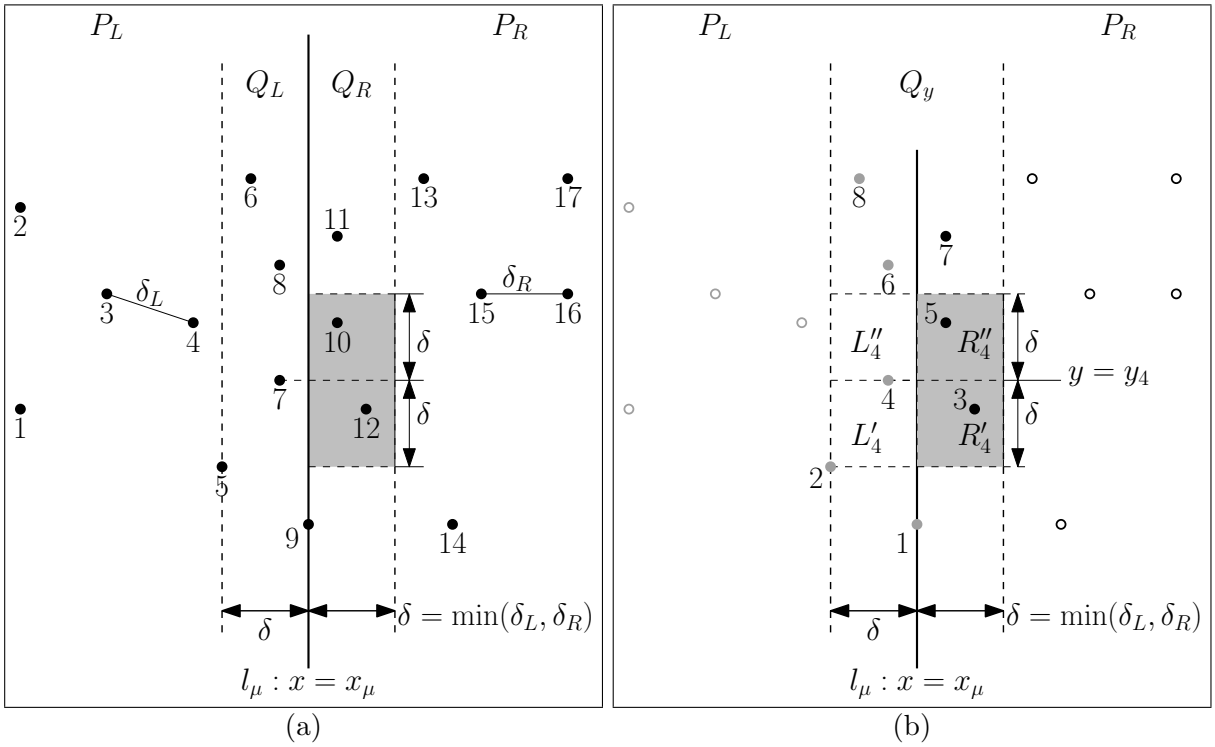


Figure 1.1: Demonstration on a small point set P having 17 points. (a) Strips $Q_L \subseteq P_L$ and $Q_R \subseteq P_R$. (b) Points and their indices in $Q_y = Q_L \cup Q_R$, sorted on y-coordinates. Points from Q_L are stored as GRAY points and those from Q_R as BLACK points, in Q_y.

1.2.1 Time complexity

Step 1: $O(n \log n)$, Step 2: $O(1)$, Step 3: $2 \times T(n/2)$, Step 4: $O(1)$, Step 5: $O(n)$.

Step 6(a): The four BLACK points following p_L in Q_y would lie in the $\delta \times \delta$ square box, namely R'', or/and above. (In Fig. 1.1b, there is one BLACK point in R''_4 corresponding to Point 4.) Also, there can be at most three other GRAY points in the $\delta \times \delta$ square box L'' containing p_L. (In Fig. 1.1b, there is no other GRAY point in L''_4.)

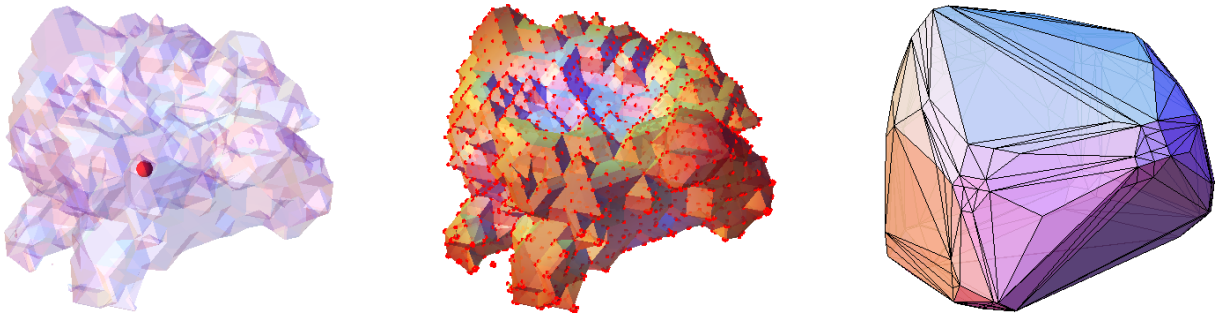
In effect, there can be at most seven other (three GRAY and four BLACK) points in $R'' \cup L''$. Hence, we have to *see at most seven points* following p_L to compute the distances of *four* BLACK points following p_L in Q_y.

Similar arguments apply also for computing the distances of *four* BLACK points preceding p_L in Q_y.

So, Step 6(a) (and hence Steps 6(b) and 6(c)) needs $O(1)$ time for each GRAY point $p_L \in Q_y$, thereby requiring $O(n)$ time for all GRAY points of Q_y.

Hence, for Steps 2–7, the time complexity is $T(n) = 2T(n/2) + O(n) = O(n \log n)$, which subsumes the time complexity of sorting (Step 1), and hence produces the overall time complexity as $\boxed{T(n) + O(n \log n) = O(n \log n)}$.

1.3 Convex Hull



The ***convex hull*** of a (2D or 3D) point set P is the *smallest convex set*[1] that contains all points of P. For a 2D set P, the convex hull may be visualized as the taut polygon formed by a rubber band stretched around the points of P. Finding the convex hull of a finite set of points is one of the fundamental problems of computational geometry.

Convex hull finds interesting applications in pattern recognition, image processing, computer graphics, statistics, GIS, and static code analysis by abstract interpretation. Hence, numerous algorithms have been proposed over the years for computing the convex hull of a finite set of 2D or 3D points, with various computational complexities. The algorithm given below is based on divide-and-conquer approach, proposed by Preparata and Hong in 1977.[2]

1. If $|P| \leqslant 3$, then compute $\mathcal{C}(P)$ in $O(1)$ time and return.
2. Partition P into P_L and P_R using the median of x-coordinates of points in P.
3. Recursively compute $\mathcal{C}(P_L)$ and $\mathcal{C}(P_R)$ for P_L and P_R respectively.
4. Merge $\mathcal{C}(P_L)$ and $\mathcal{C}(P_R)$ to get $\mathcal{C}(P)$ by computing their lower and upper tangents and deleting all the vertices of $\mathcal{C}(P_L)$ and $\mathcal{C}(P_R)$ (except the vertex points of tangency) lying between these two tangents[3](Fig. 1.2).

   Finding the Lower Tangent:

   Let a be the rightmost vertex of $\mathcal{C}(P_L)$ and b be the leftmost vertex of $\mathcal{C}(P_R)$.

   (a) while ab is not a lower tangent for $\mathcal{C}(P_L)$ and $\mathcal{C}(P_R)$

   i. while ab is not a lower tangent[4] to $\mathcal{C}(P_L)$
   $a \leftarrow a + 1$ (move a clockwise)

   ii. while ab is not a lower tangent to $\mathcal{C}(P_R)$
   $b \leftarrow b - 1$ (move b counterclockwise)

   (b) return ab

   The upper tangent can be obtained in a similar manner.

[1] In Euclidean space, a set or an object is *convex* if for every pair of points within the object, every point on the straight line segment joining them is also within the object.

[2] Franco P. Preparata and S. J. Hong. Convex Hulls of Finite Sets of Points in Two and Three Dimensions, *Commun. ACM*, **20**(2):87–93, 1977.

[3] A ***lower (upper) tangent*** is the straight line that touches the two hulls such that all the vertices of the two hulls lie above (below) the tangent.

[4] ab is not a lower tangent if the vertex $a+1$ of $\mathcal{C}(P_L)$, lying next to a, lies to the left of the ray/vector directed from b to a.

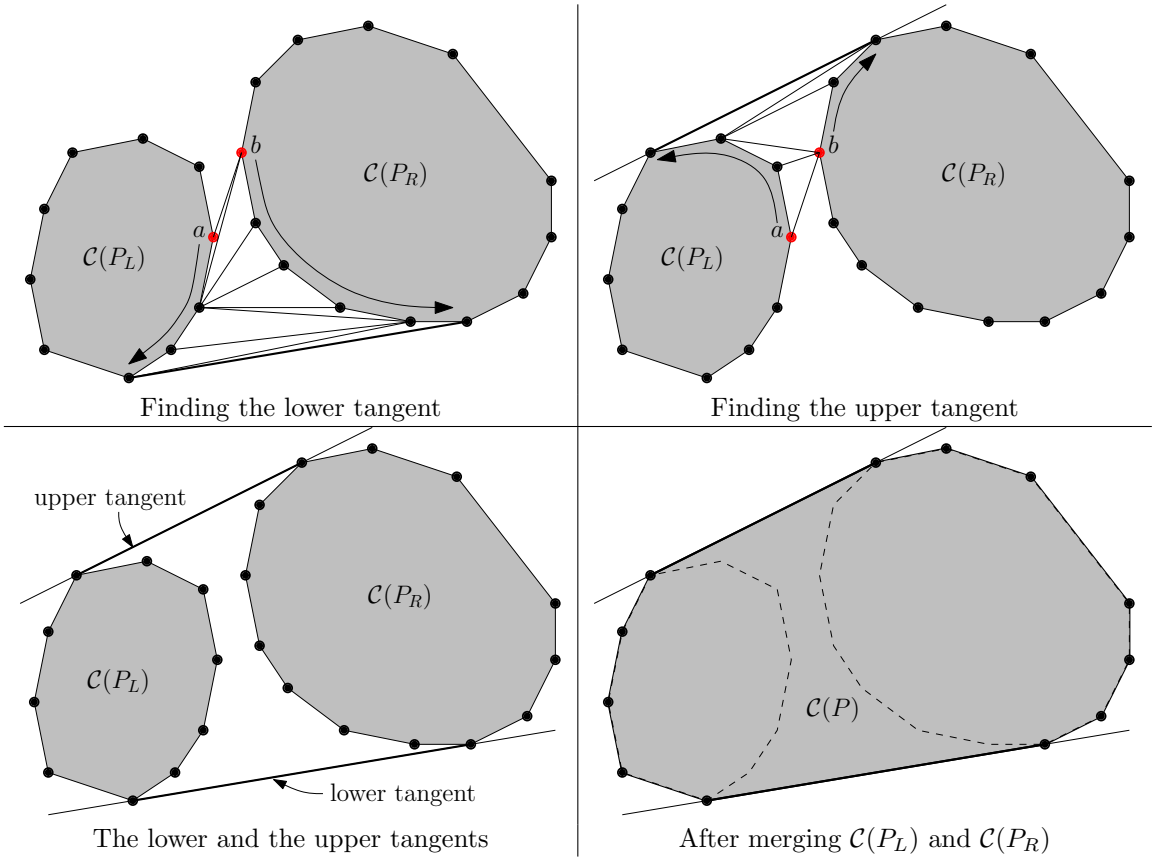


Figure 1.2: Divide-and-conquer approach to find the convex hull $\mathcal{C}(P)$ of P: Recursively finding $\mathcal{C}(P)$ using the convex hulls $\mathcal{C}(P_L)$ and $\mathcal{C}(P_R)$ corresponding to P_L and P_R, and their lower and upper tangents.

1.3.1 Time complexity

While finding a tangent, each vertex on each hull will be checked at most once (e.g., $a+1$ will be checked only once—whether it lies to the left of the directed ray $\overrightarrow{ba}$). Hence, the runtime to find the two tangents is $O(|\mathcal{C}(P_L)| + |\mathcal{C}(P_R)|) = O(n)$.

Hence, Step 4 (merging) of the algorithm needs $O(n)$ time. Step 1 needs $O(1)$ time and Step 2 needs $O(n)$ time for median-based partitioning. Step 3 is recursive. Thus, $T(n) = 2T(n/2) + O(n)$, which solves to $O(n \log n)$.

Chapter 2

Dynamic Programming

What is a cynic? A man who knows the price of everything and the value of nothing.

— **Oscar Wilde**

Dynamic programming (DP) solves a complex problem by breaking it into simpler overlapping subproblems, when the problem has an optimal substructure. It seeks to solve each subproblem only once, thus reducing the computation. Once the solution to a given subproblem has been computed, it is stored or ***memo-ized***, so that the next time the same solution is needed, it is simply looked up.

2.1 Elements of Dynamic Programming

Optimal substructure: An optimal solution to the problem contains optimal solutions to subproblems (as in greedy algorithms).

Overlapping subproblems: A sub-subproblem P' has to be solved repeatedly to solve two or more subproblems whose intersection is P'. Hence, the problem contains a recursive nature, which often creates an illusion of exponential possibilities.

Example problems: Matrix-chain multiplication; Longest common subsequence; 0-1 Knapsack problem; Optimal triangulation of a convex polygon (each triangle has some weight, e.g., $w(a, b, c) = |ab| + |bc| + |ca|$); Denomination problem; All-pair shortest paths; Weighted interval scheduling.

Note: If a problem can be solved by combining optimal solutions to non-overlapping subproblems, then it is ***divide and conquer***, e.g., quicksort.

2.2 Weighted Interval Scheduling

Given a set of n intervals, $R = \{I_1, I_2, \ldots, I_n\}$, each $I_j = [a_j, b_j]$ having a real-valued weight w_j, find the largest set of intervals $S \subseteq R$ such that no two intervals in S overlap each other and the sum of weights of the intervals in S is maximized.

Relation with Scheduling: Each interval I_j can be thought as a process, with starting time as a_j and finishing time as b_j. Hence, for ease of understanding, S is considered as a sequence of processes in which a new process can start only when an existing process is finished. So we sort the intervals of R in nondecreasing order of their finishing times to get $b_1 \leqslant b_2 \leqslant \ldots \leqslant b_n$. Henceforth we explain the algorithm on sorted R.

Define $p_j = i$ where I_i lies immediately precedes I_j, i.e., $a_j \geqslant \max\{b_i : I_i \in R\}$.
Define $S(j)$ as the optimal solution for $I_1, \ldots, I_j$, and $W(j)$ as the sum of weights of the intervals in $S(j)$.

Optimal substructure: The observation is as follows.

(i) if $W(p_j) + w_j < W(j-1)$, then $I_j \notin S(j)$, $S(j) = S(j-1)$, and $W(j) = W(j-1)$.

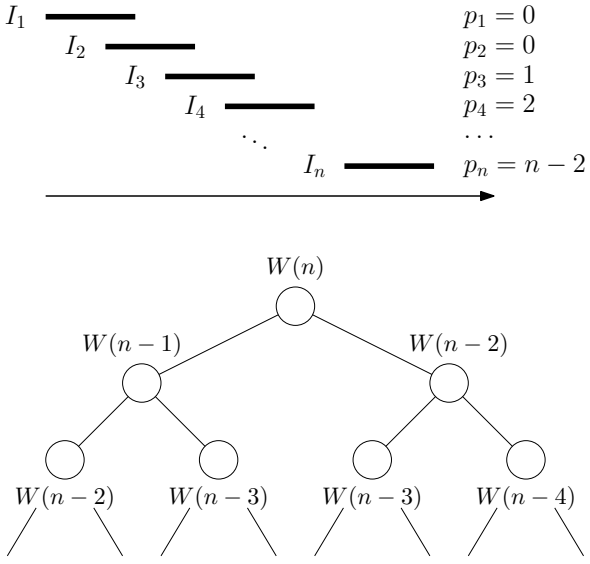
(ii) if $W(p_j) + w_j \geqslant W(j-1)$, then $I_j \in S(j)$, $S(j) = S(p_j) \cup \{I_j\}$, and $W(j) = W(p_j) + w_j$.

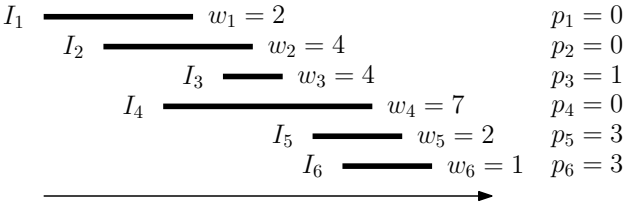
Overlapping subproblems:

$$W(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\left(W(p_j) + w_j, W(j-1)\right) & \text{otherwise} \end{cases} \tag{2.1}$$

The algorithm: Algorithm 1 is based on direct implementation of Eqn. 2.1. Figure 2.1 shows a typical example where computation of $W(n)$ recursively invokes $W(n-2)$ and $W(n-1)$, and hence follows the pattern of Fibonacci sequence. The recursion tree has therefore an exponential space complexity, which makes the algorithm inefficient. This is avoided by ***memoization***, which is a technique of saving the computed values that will be needed during the process of recursion. Algorithm 2 is based on such memoization. It should be run for $j = 1$ to n, with W implemented as an array, initialized with $W(j) = 0$ for $j = 0, 1, \ldots, n$. For each value of j, time required is $O(1)$, thus getting a linear time complexity in total. Algorithm 3 is the most simplified version. It is an iterative algorithm and free of memoization! It is so designed as Step 6 recurses always once for each j, just because $W[p_j]$ and $W[j-1]$ are computed before computing $W[j]$. Figure 2.1 shows a demo.

Time complexity: Sorting needs $O(n \log n)$ time and finding all p_js for $j = 1, \ldots, n$ from the sorted list requires $n \cdot O(\log n) = O(n \log n)$ time. Step 3 of Algorithm 3 needs constant time, since $W[p_j]$ and $W[j-1]$ are already computed. Hence, computation of $W[n]$ requires $O(n)$ time.





run	j = 0	1	2	3	4	5	6	$S(j)$
1	0	2						$\{I_1\}$
2	0	2	4					$\{I_2\}$
3	0	2	4	6				$\{I_1, I_3\}$
4	0	2	4	6	7			$\{I_4\}$
5	0	2	4	6	7	8		$\{I_1, I_3, I_5\}$
6	0	2	4	6	7	8	8	$\{I_1, I_3, I_5\}$

Figure 2.1: **Left:** An example ($w_1 = w_2 = \ldots = w_n = 1$) that indeed needs exponential time and space complexities for Algorithm 1 (direct implementation of Eqn. 2.1). **Right:** An example demonstrating computation of $W(j)$ by Algorithm 3. For jth run to compute $W(j)$ and $S(j)$, $W(p_j)$ has been highlighted in gray.

Algorithm 1: An inefficient algorithm that directly implements Eqn. 2.1.

```
if j = 0 then
    return 0
else
    return max(W(p_j) + w_j, W(j − 1))
```

Algorithm 2: An $O(n)$ algorithm that implements Eqn. 2.1 through memoization.

```
if j = 0 then
    return 0
else if W(j) ≠ 0 then
    return W(j)
else
    return W(j) ← max(W(p_j) + w_j, W(j − 1))
```

Algorithm 3: An $O(n)$ iterative algorithm, requiring no memoization!

```
W[0] ← 0
for j ← 1, ..., n do
    W(j) ← max(W[p_j] + w_j, W[j − 1])
return W[n]
```

2.3 0-1 (Binary) Knapsack Problem

Given a set $A = \{a_i : i = 1, 2, \ldots, n\}$ where each a_i has benefit b_i and (a positive integer) weight w_i. Given a knapsack whose weight carrying capacity is k. The problem is to put some or all items from A into the knapsack without exceeding its capacity such that the total benefit is maximized.

Define $A_i = \{a_1, a_2, \ldots, a_i\}, 1 \leqslant i \leqslant n$,
and let $B[i, w]$ = maximum total benefit over all possible subsets $A_j \subseteq A_i$ such that $\sum_{a_j \in A_j} w_j = w$.

Observation: If $w_i > w$, then a_i can never be in the maximum-benefit solution A_j corresponding to $B[i, w]$. Otherwise, there may arise two cases: In case $a_i \in A_j$, we have $B[i, w] = B[i-1, w-w_i] + b_i$; and in case $a_i \notin A_j$, we have $B[i, w] = B[i-1, w]$. Thus,

$$B[i, w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ B[i-1, w] & \text{if } w_i > w \\ \max\{B[i-1, w], B[i-1, w-w_i] + b_i\} & \text{if } w_i \leqslant w. \end{cases} \tag{2.2}$$

The elements of DP are evident from Eqn. 2.2. See Algorithm 4.

Algorithm 4: 0-1 Knapsack

```
1 for w ← 0, 1, ..., k do
2     B[0, w] ← 0
3 for i ← 1, 2, ..., n do
4     for w ← 0, 1, ..., k do
5         if w ⩾ w_i then
6             B[i, w] ← max(B[i − 1, w], B[i − 1, w − w_i] + b_i)
7         else
8             B[i, w] ← B[i − 1, w]
```

Time complexity: Steps 3–8 dominate over the first two steps (initialization) in runtime. The inner **for** loop (starting from Step 4) takes $O(k)$ time and the outer **for** loop (starting from Step 3) iterates for n times. Hence, the overall time complexity is $O(n) \times O(k) = O(nk)$.[1]

Example:

Let $n = 4$, $k = 6$.

i	1	2	3	4
b_i	7	9	6	8
w_i	2	4	7	3

Iterations:

	w	0	1	2	3	4	5	6
$B[w]$	initial	0	0	0	0	0	0	0
	$i = 1$	0	0	7	7	7	7	7
	$i = 2$	0	0	7	7	9	9	16
	$i = 3$	0	0	7	7	9	9	16
	$i = 4$	0	0	0	8	9	15	16

$\Rightarrow$ max benefit = 16.

[1] Such an algorithm is called a ***pseudo-polynomial algorithm***, since its polynomial time complexity is not only dependent on the input size, i.e., n, but also on k, which is just an input value. In fact, 0-1 knapsack problem is considered as an ***NP-complete problem***.

2.4 Fractional Knapsack Problem

If we are allowed to take a fraction of any item $a_i \in A$ while maximizing the total benefit without exceeding the knapsack capacity, then it's ***fractional knapsack problem***. Such a problem is solvable by ***greedy approach*** as follows. Compute unit-benefit $c_i = b_i/w_i$ for each a_i, and arrange a_i's in decreasing/non-increasing order of c_i's. Now, select the items starting from the first item of the sorted list, ending with a fraction f_j of some item, a_j, such that all other items selected so far (possibly excepting a_j, if $f_j < 1$) are fully selected. Due to sorting, its time complexity $= O(n \log n)$.

Clearly, fractional knapsack problem is much easier than 0-1 knapsack.

Example: Let $n = 4$, $k = 6$. We arrange the items in decreasing order of c_i's as follows.

i	1	2	3	4
b_i	12	12	6	7
w_i	2	3	2	7
c_i	6	4	3	1

Output: $f_1 = 1, f_2 = 1, f_3 = 1/2, f_4 = 0$, and total benefit $= 12 + 12 + 3 = 27$.

2.5 Longest common subsequence (LCS)

Given two sequences, $A = \langle a_i : i = 1, \ldots, m\rangle$ and $\langle b_j : j = 1, \ldots, n\rangle$, find the longest sequence $C = \langle c_k : k = 1, \ldots, p\rangle$ such that

(i) each c_k matches with some $a \in A$ and with some $b \in B$;

(ii) all the characters (if any) of C preceding c_k match with some $k-1$ characters of A preceding a and with some $k-1$ characters of B preceding b.

$C_p = LCS(A_m, B_n)$ (or, simply $C = LCS(A, B)$) is called the ***longest common subsequence*** of A and B.

Example: $A = \langle$algorithms$\rangle$, $B = \langle$allgorhythm$\rangle$: $C = \langle$`algorthm`$\rangle$.

Optimal substructure: The observation is as follows.

(i) if $a_m = b_n$, then $c_p = a_m = b_n$ and $C_{k-1} = LCS(A_{m-1}, B_{n-1})$.

(ii) if $a_m \neq b_n$, then $c_p \neq x_m$ implies $C_p = LCS(A_{m-1}, B_n)$; else $c_p \neq y_n$ implying $C_p = LCS(A_m, B_{n-1})$.

Overlapping subproblems: Let $|LCS(A_i, B_j)|$ be the length of $LCS(A_i, B_j)$, where $A_i = \langle a_1, a_2, \ldots, a_i\rangle$ and $B_j = \langle b_1, b_2, \ldots, b_j\rangle$. Then we have

$$|LCS(A_i, B_j)| = \begin{cases} 0 & \text{if } i = 0 \vee j = 0 \\ |LCS(A_{i-1}, B_{j-1})| + 1 & \text{if } i > 0 \wedge j > 0 \wedge a_i = b_j \\ \max\left(|LCS(A_i, B_{j-1})|, |LCS(A_{i-1}, B_j)|\right) & \text{if } i > 0 \wedge j > 0 \wedge a_i \neq b_j \end{cases} \quad (2.3)$$

		a	l	g	o	r	i	t	h	m
g	0	0	1	1	1	1	1	1	1
l	0	1	1	1	1	1	1	1	1
o	0	1	1	2	2	2	2	2	2
b	0	1	1	2	2	2	2	2	2
a	1	1	1	2	2	2	2	2	2
l	1	2	2	2	2	2	2	2	2
i	1	2	2	2	2	3	3	3	3
s	1	2	2	2	2	3	3	3	3
m	1	2	2	2	2	3	3	3	4

Figure 2.2: LCS of two strings, '`algorithm`' and '`globalism`'.

2.6 Matrix-chain Multiplication

Given a chain of matrices $A_1A_2\cdots A_n$, the problem is to obtain its fully ***parenthesized form*** such that the number of scalar multiplications is minimized. Apparently, the problem seems to be have exponential complexity as it relates to ***Catalan number***, as follows.

Number of parenthesizations:

$$\begin{aligned} P(n) &= \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geqslant 2 = \textbf{\textit{Catalan number}}\ C(n-1) \end{cases} \\ \text{where} \quad C(n) &= \frac{1}{n+1}\binom{2n}{n} = \Omega\left(\frac{4^n}{n^{3/2}}\right). \end{aligned} \tag{2.4}$$

Using DP, we can solve it in low-order polynomial time. The DP-based algorithm is built on the following observation, which explains the two elements of DP.

Optimal substructure & overlapping subproblems: Parenthesization of $A_1A_2\cdots A_k$ $(k < n)$ within the optimal parenthesization of $A_1A_2\cdots A_n$ must be an optimal parenthesization of $A_1A_2\cdots A_k$; and similar for $A_{k+1}\cdots A_n$.

Thus, the minimum number of scalar multiplications for $A_iA_{i+1}\ldots A_j$ is

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \leqslant k < j}(m[i,k] + m[k+1,j] + p_{i-1}p_kp_j) & \text{if } i < j; \end{cases}$$

where each A_i has p_{i-1} rows and p_i columns.

Algorithm 5: Matrix chain multiplication

```
1 Initialize m[i,i] = 0 for i = 1,2,...,n
2 for h ← 1,2,...,n−1 do
3     for i ← 1,2,...,n−h do
4         j ← i+h, m[i,j] ← ∞
5         for k ← i,...,j−1 do
6             q ← m[i,k] + m[k+1,j] + p_{i−1} p_k p_j
7             if q < m[i,j] then
8                 m[i,j] ← q, s[i,j] ← k
9 return m and s
```

Time complexity: The outermost **for** loop (Step 2) has $n-1$ iterations, and the next **for** loop (Step 3 has $n-h$ iterations for hth iteration of the outermost **for** loop. The innermost **for** loop (Step 5) has $j-i=h$ iterations. Thus, total number of computations is $O(n-1)\cdot 1+O(n-2)\cdot 2+O(n-3)\cdot 3+\ldots+O(n)\cdot(n-1) = O\big((n-1)+2(n-2)+3(n-3)+\ldots+(n-(n-1))(n-1)\big) = O(n^3)$.

Example: Minimize the number of scalar multiplications to compute $A_1 A_2 \cdots A_6$ for:

matrix	dimension
A_1	20×30
A_2	30×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

In the given problem, $j - i = 1, 2, \ldots, 5$.
We start with $h = j - i = 1$ and finally end at $j - i = 5$ to reach the final solution, i.e., $m[1, 6]$.

1. $j - i = 1$:
   (a) $m[1,2] = p_0 p_1 p_2 = 20 \times 30 \times 15 = 9000$
   (b) $m[2,3] = p_1 p_2 p_3 = 30 \times 15 \times 5 = 2250$
   (c) $m[3,4] = p_2 p_3 p_4 = 15 \times 5 \times 10 = 750$
   (d) $m[4,5] = p_3 p_4 p_5 = 5 \times 10 \times 20 = 1000$
   (e) $m[5,6] = p_4 p_5 p_6 = 10 \times 20 \times 25 = 5000$
2. $j - i = 2$:
   (a) $m[1,3] = \min \left\{ \begin{array}{lll} m[1,2] + p_0 p_2 p_3 & = 9000 + 1500 & = 10500 \\ m[2,3] + p_0 p_1 p_3 & = 2250 + 3000 & = 5250 \end{array} \right\} = 5250 (k = 1)$
   (b) $m[2,4] = \min \left\{ \begin{array}{lll} m[2,3] + p_1 p_3 p_4 & = 2250 + 1500 & = 3750 \\ m[3,4] + p_1 p_2 p_4 & = 750 + 4500 & = 5250 \end{array} \right\} = 3750 (k = 3)$
   (c) $m[3,5] = \min \left\{ \begin{array}{lll} m[3,4] + p_2 p_4 p_5 & = 750 + 3000 & = 3750 \\ m[4,5] + p_2 p_3 p_5 & = 1000 + 1500 & = 2500 \end{array} \right\} = 2500 (k = 3)$
   (d) $m[4,6] = \min \left\{ \begin{array}{lll} m[4,5] + p_3 p_5 p_6 & = 1000 + 2500 & = 3500 \\ m[5,6] + p_3 p_4 p_6 & = 5000 + 1250 & = 6250 \end{array} \right\} = 3500 (k = 5)$
3. $j - i = 3$:
   (a) $m[1,4] = \min \left\{ \begin{array}{lll} m[1,3] + p_0 p_3 p_4 & = 5250 + 1000 & = 6250 \\ m[2,4] + p_0 p_1 p_4 & = 3750 + 6000 & = 9750 \\ m[1,2] + m[3,4] + p_0 p_2 p_4 & = 9000 + 750 + 3000 & = 12750 \end{array} \right\} = 6250 (k = 3)$
   (b) $m[2,5] = \min \left\{ \begin{array}{lll} m[2,4] + p_1 p_4 p_5 & = 3750 + 6000 & = 9750 \\ m[3,5] + p_1 p_2 p_5 & = 2500 + 9000 & = 11500 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 & = 2250 + 1000 + 3000 & = 6250 \end{array} \right\} = 6250 (k = 3)$
   (c) $m[3,6] = \min \left\{ \begin{array}{lll} m[3,5] + p_2 p_5 p_6 & = 2500 + 7500 & = 10000 \\ m[4,6] + p_2 p_3 p_6 & = 3500 + 1875 & = 5375 \\ m[3,4] + m[5,6] + p_2 p_4 p_6 & = 750 + 5000 + \times & = \times \end{array} \right\} = 5375 (k = 3)$
4. $j - i = 4$:
   (a) $m[1,5] = \min \left\{ \begin{array}{lll} m[1,4] + p_0 p_4 p_5 & = 6250 + 4000 & = 10250 \\ m[2,5] + p_0 p_1 p_5 & = 6250 + 12000 & = 18250 \\ m[1,3] + m[4,5] + p_0 p_3 p_5 & = 5250 + 1000 + 2000 & = 8250 \\ m[1,2] + m[3,5] + p_0 p_2 p_5 & = 9000 + 2500 + 6000 & = 17500 \end{array} \right\} = 8250 (k = 3)$
   (b) $m[2,6] = \min \left\{ \begin{array}{lll} m[2,5] + p_1 p_5 p_6 & = 6250 + 15000 & = 21250 \\ m[3,6] + p_1 p_2 p_6 & = 5375 + 11250 & = 16625 \\ m[2,3] + m[4,6] + p_1 p_3 p_6 & = 2250 + 3750 + 3500 & = 9500 \\ m[2,4] + m[5,6] + p_1 p_4 p_6 & = 3750 + 5000 + 7500 & = 16250 \end{array} \right\} = 9500 (k = 3)$
5. $j - i = 5$:

$$\text{(a)}\quad m[1,6] = \min \left\{ \begin{array}{lll} m[1,5] + p_0p_5p_6 & = 8250 + 10000 & = 18250 \\ m[2,6] + p_0p_1p_6 & = 9500 + 15000 & = 24500 \\ m[1,2] + m[3,6] + p_0p_2p_6 & = 9000 + 5375 + 7500 & = 21875 \\ m[1,3] + m[4,6] + p_0p_3p_6 & = 5250 + 3500 + 2500 & = 11250 \\ m[1,4] + m[5,6] + p_0p_4p_6 & = 6250 + 5000 + 5000 & = 16250 \end{array} \right\} = 11250(k = 3)$$

Table (value of k is shown parenthesized with the corresponding m):

	1	2	3	4	5	6
1	0	9000 (1)	5250 (1)	6250 (3)	8250 (3)	11250 (3)
2		0	2250 (2)	3750 (3)	6250 (3)	9500 (3)
3			0	750 (3)	2500 (3)	5375 (3)
4				0	1000 (4)	3500 (5)
5					0	5000 (5)
6						0

Since $k[1,6] = 3$, we have the parenthesization as follows: $(A_1A_2A_3)(A_4A_5A_6)$
$= (A_1(A_2A_3))((A_4A_5)A_6)$, since $k[1,3] = 1$ and $k[4,6] = 5$.

2.7 Matrix Multiplication

Let A and B be two $n \times n$ square matrices, where n is an exact power of 2. The problem is to obtain $C = AB$ with minimum/reduced computation[1]. The steps of Strassen's Algorithm are as follows:

1. Decompose A, B, C, each in the form of four $\frac{n}{2} \times \frac{n}{2}$ matrices, as follows.

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & g \\ f & h \end{bmatrix}, \tag{2.5}$$

   where

$$r = ae + bf,\ s = ag + bh,\ t = ce + df,\ u = cg + dh. \tag{2.6}$$

2. Compute the following seven matrices, each of size $\frac{n}{2} \times \frac{n}{2}$ [Additions[2] $= 5\left(\frac{n}{2}\right)^2$]:

$$A_1 = a,\ A_2 = a + b,\ A_3 = c + d,\ A_4 = d,\ A_5 = a + d,\ A_6 = b - d,\ A_7 = a - c. \tag{2.7}$$

3. Compute the following seven matrices, each of size $\frac{n}{2} \times \frac{n}{2}$ [Additions $= 5\left(\frac{n}{2}\right)^2$]:

$$B_1 = g - h,\ B_2 = h,\ B_3 = e,\ B_4 = f - e,\ B_5 = e + h,\ B_6 = f + h,\ B_7 = e + g. \tag{2.8}$$

4. Recursively compute $P_i = A_i B_i$ for $i = 1, \ldots, 7$. [Time $= 7T(\frac{n}{2})$]:

5. Compute r, s, t, u (Strassen's intuition!) from P_is as follows [Additions $= 8\left(\frac{n}{2}\right)^2$]:

$$\begin{aligned} r &= -P_2 + P_4 + P_5 + P_6 \\ &= -(a+b)h + d(f-e) + (a+d)(e+h) + (b-d)(f+h) = ae + bf, \\ s &= P_1 + P_2 = a(g-h) + (a+b)h = ag + bh, \\ t &= P_3 + P_4 = (c+d)e + d(f-e) = ce + df, \\ u &= P_1 - P_3 + P_5 - P_7 \\ &= a(g-h) - (c+d)e + (a+d)(e+h) - (a-c)(e+g) = cg + dh \end{aligned} \tag{2.9}$$

Time complexity: Number of scalar additions (Steps 2,3,5) $= (5+5+8)\left(\frac{n}{2}\right)^2 = 18\left(\frac{n}{2}\right)^2 = O(n^2)$. Step 4 needs $7T(\frac{n}{2})$ time to compute $P_1, \ldots, P_7$. Hence,

$$\begin{aligned} T(n) &= 7T\left(\frac{n}{2}\right) + O(n^2) \\ &= O(n^{\log_2 7}) \quad \text{[by Master Method]} \\ &= O(n^{2.81}). \end{aligned}$$

[1] Although this problem does not require DP, it is put in this chapter, as matrix-chain multiplication is discussed in Sec. 2.6.

[2] (Scalar) addition and subtraction are equivalent operations.

2.8 All-pair shortest paths

Let $G(V, E)$ be a *weighted, directed* graph with a weight function $w : E \mapsto \mathbb{R}$. For every pair of vertices $u, v \in V$, we have to find out the shortest path from u to v. Recall that Dijkstra's algorithm takes $O(E \log V)$ time, which is $O(V^2 \log V)$, when $E = O(V^2)$. Hence on applying Dijkstra's algorithm with each vertex of V as source vertex, we take $O(V^3 \log V)$ time for a dense graph. Further, if some edge has negative weight, then Dijkstra's algorithm cannot be used. We can use dynamic programming to have efficient algorithms. We assume that there is *no cycle with negative weight*, although negative-weight edges are permitted.

Input: Adjacency matrix $W = (w_{ij})$ of size $n \times n$, where $n = |V|$ and

$$w_{ij} = \begin{cases} 0 & \text{if } i = j, \\ \text{weight of } (i,j) & \text{if } i \neq j \text{ and } (i,j) \in E, \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E. \end{cases}$$

Output: (i) $D = (d_{ij})$ of size $n \times n$, where $d_{ij} =$ shortest path length from i to j. (ii) Predecessor matrix $\Pi = (\pi_{ij})$, where $\pi_{ij} = nil$ if $i = j$ or no path exists from i to j; otherwise π_{ij} is some predecessor of j on a shortest path from i to j.

2.8.1 Recursive solution

Let $d_{ij}^{(m)}$ denote the minimum weight of *any path* $p_{ij}^{(m)}$ from i to j that contains *at most* m edges. So, for $m = 0$,

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j, \\ \infty & \text{if } i \neq j. \end{cases}$$

For $m \geqslant 1$, we consider each predecessor k of j to get

$$\begin{aligned} d_{ij}^{(m)} &= \begin{cases} d_{ij}^{(m-1)} & \text{if } k \notin p_{ij}^{(m)} \\ \min\limits_{1 \leqslant k \leqslant n} \left\{ d_{ik}^{(m-1)} + w_{kj} \right\} & \text{if } k \in p_{ij}^{(m)} \end{cases} \\ &= \min \left\{ d_{ij}^{(m-1)}, \min_{1 \leqslant k \leqslant n} \left\{ d_{ik}^{(m-1)} + w_{kj} \right\} \right\}. \end{aligned}$$

Since $\left\{ d_{ik}^{(m-1)} + w_{kj} \right\}$ for $1 \leqslant k \leqslant n$ includes $d_{ij}^{(m-1)} + w_{jj} = d_{ij}^{(m-1)}$ also, we can write

$$d_{ij}^{(m)} = \min_{1 \leqslant k \leqslant n} \left\{ d_{ik}^{(m-1)} + w_{kj} \right\}.$$

Note: As any shortest path from any vertex i to any vertex j has at most $n-1$ edges, we have $d_{ij}^{(n-1)} = d_{ij}^{(n)} = d_{ij}^{(n+1)} = \ldots$.

Extend-Shortest-Paths (Algorithm 6) needs $O(n^3)$ time for 3 nested **for** loops. It should be called $n-2$ times, starting from $D^{(1)} = W$ and ending at $D^{(n-2)}$, so as to get the solution as $D^{(n-1)}$ (see Algo 7). Hence, total time is $O(n^4)$, which can be reduced to $O(n^3 \log n)$ as follows. Compute $D^{(2m)}$ by calling Extend-Shortest-Paths$(D^{(m)}, D^{(m)})$, for $m = 1, 2, \ldots, \lceil \log(n-1) \rceil$.

Algorithm 6: EXTEND-SHORTEST-PATHS(D, W).

Input: $D = (d_{ij}^{(m-1)}), W$
Output: $D' = (d_{ij}^{(m)})$

```
1 for i ← 1 to n do
2     for j ← 1 to n do
3         D'[i][j] ← ∞
4         for k ← 1 to n do
5             D'[i][j] ← min{D'[i][j], D[i][k] + W[k][j]}
```

Algorithm 7: ALL-PAIR-SHORTEST-PATHS-SLOW(G, W).

Input: G, W
Output: $D^{(n-1)}$

```
1 D^(1) ← W
2 for m ← 2 to n − 1 do
3     D^(m) ← EXTEND-SHORTEST-PATHS(D^(m−1), W)
4 return D^(n−1)
```

Refinement: Let's take $d_{ij}^{(2m)}$. It is the weight of the shortest path from i to j with *at most* $2m$ edges. We can *always* split such a shortest path p as two (shortest) sub-paths: p_1 from i to k and p_2 from k to j, each consisting of at most m edges. So, we can write

$$d_{ij}^{(2m)} = \min_{1 \leqslant k \leqslant n} \left\{ d_{ik}^{(m)} + d_{kj}^{(m)} \right\}.$$

Note: In the above equation, the range (i.e., $[1, n]$) of k allows us such a formulation.

Thus, we can compute $D^{(2m)}$ only for $m = 1, 2, 2^2, \ldots$ until we get $D^{(n')}$, where $n' \geqslant n - 1$. This ensures $\Theta(\log n)$ iterations for Step 2 of Algo 7, which improves the time complexity to $O(n^3 \log n)$.

2.8.2 Floyd-Warshall Algorithm

Observation: Let $V = \{1, 2, \ldots, n\}$.
Consider any subset $V_k = \{1, 2, \ldots, k\}$.
For any $(i, j) \in V \times V$, consider all paths $\{p_{ij}^{(k)}\}$ from i to j whose intermediate vertices are only from V_k, and let $d_{ij}^{(k)}$ be the minimum weight of all these paths. Then there arise two cases:

1. k is not an intermediate vertex in $p_{ij}^{(k)}$: $d_{ij}^{(k)} = d_{ij}^{(k-1)}$.
2. k is an intermediate vertex in $p_{ij}^{(k)}$: $d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$ (Fig. 2.3).

So,

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0, \\ \min\left\{ d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right\} & \text{if } k \geqslant 1. \end{cases}$$

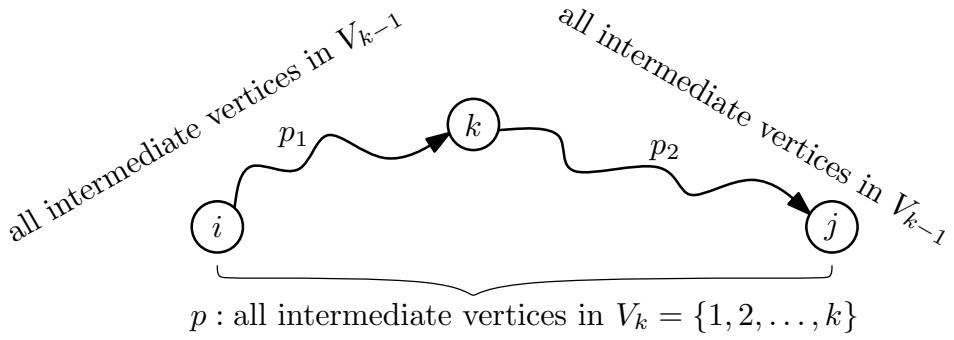


Figure 2.3: Floyd-Warshall Algorithm: Decomposition of $p := p_{ij}^{(k)}$ into p_1 and p_2 when $k \in p$.

Algorithm 8: FLOYD-WARSHALL(G, W).

```
1 D^(0) ← W
2 for k ← 1 to n do
3     for i ← 1 to n do
4         for j ← to n do
5             d_ij^(k) ← min{ d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1) }
6 return D^(n)
```

Time complexity: $O(n^3)$, space complexity: $O(n^2) \times O(n) = O(n^3)$.

Refinement: Just remove the superscripts in Algo 8. Then we have a single matrix D which finally contains the solution. Space requirement $= O(n^2)$—an improvement! The refined algorithm is given below.

Algorithm 9: FLOYD-WARSHALL-REFINED(G, W).

```
1 D ← W
2 for k ← 1 to n do
3     for i ← 1 to n do
4         for j ← to n do
5             D[i][j] ← min{D[i][j], D[i][k] + D[k][j]}
6 return D
```

Algorithm 10: TRANSITIVE-CLOSURE(G, W).

```
1  for i ← 1 to n do
2      for j ← to n do
3          if W[i][j] ≠ ∞ then
4              G*[i][j] ← 1
5          else
6              G*[i][j] ← 0
7  for k ← 1 to n do
8      for i ← 1 to n do
9          for j ← to n do
10             G*[i][j] ← G*[i][j] ∨ (G*[i][k] ∧ G*[k][j])
11 return G*
```

2.8.3 Transitive Closure

For a digraph $G(V, E)$, the transitive closure $G^* = (V, E^*)$,
where $E^* = \{(i, j) :$ there is a path from i to j in $G\}$.

We can represent G^* as a binary matrix of size $n \times n$, where $n = |V|$, such that $G^*[i][j] = 1$ if and only if $(i, j) \in E^*$. There are two ways to find G^*.

Way 1: Major steps are as follows:

1. Assign weight 1 to each edge of E.
2. Run Floyd-Warshall to find D.
3. If $D[i][j] < n$, then make $G^*[i][j] = 1$; otherwise $D[i][j] = \infty$, make $G^*[i][j] = 0$.

Way 2: Let $t_{ij}^{(k)} = 1$ iff there is a path from i to j with intermediate vertices from $V_k = \{1, 2, \ldots, k\}$. Then we get a recurrence similar to $d_{ij}^{(k)}$ as follows.

$$t_{ij}^{(0)} = \begin{cases} 1 & \text{if } i = j \text{ or } (i, j) \in E, \\ 0 & \textit{otherwise.} \end{cases}$$

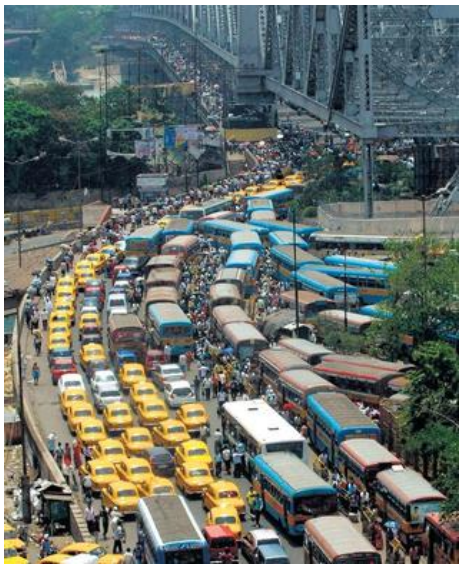
For $k \geqslant 1$,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)} \right).$$

The algorithm is given in Algo 10, which needs $O(n^3)$ time and $O(n^2)$ space.

Chapter 3

Flow Networks



If you optimize everything, you will always be unhappy.

— **Donald Knuth**

A ***flow network*** $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a ***capacity*** $c(u, v) \geqslant 0$. If $(u, v) \notin E$, then $c(u, v) = 0$. Two special vertices, namely a ***source*** s, and a ***sink*** t, are considered from V.

A ***flow*** in G is a real-valued function $f : V^2 \to \mathbb{R}$ that satisfies the following ***necessary and sufficient properties***:

1. ***Capacity constraint:*** $f(u, v) \leq c(u, v)\ \forall (u, v) \in V^2$.
2. ***Skew symmetry:*** $f(u, v) = -f(v, u)\ \forall (u, v) \in V^2$.
3. ***Flow conservation:*** $f(u, V) = \sum_{v \in V} f(u, v) = 0\ \forall u \in V \setminus \{s, t\}$.

The ***net flow*** from a vertex u to a vertex v is the quantity $f(u, v)$ that can be positive, negative, or zero. The ***value*** of f in G is defined as

$$|f| = f(s, V) = \sum_{v \in V} f(s, v). \tag{3.1}$$

Maximum-flow problem: Maximize $|f|$ for a given G.

3.1 Properties of a flow network

1. $\boxed{f(u,u) = 0 \ \forall u \in V}$

   Proof: From skew symmetry, for $v = u$, $f(u,u) = -f(u,u)$. □

2. $\boxed{f(V,u) = 0 \ \forall u \in V \setminus \{s,t\}}$

   Proof: From skew symmetry, for all $(u,v) \in V^2$, $f(u,v) = -f(v,u)$, or, $\sum_{u \in V} f(u,v) = -\sum_{u \in V} f(v,u)$, or, $f(V,v) = -f(v,V) = 0 \forall v \in V \setminus \{s,t\}$ (by flow conservation). □

Lemma 3.1.1 (Subgraph flow)

1. $\boxed{f(X,X) = 0 \text{ for } X \subseteq V}$

   Proof: From skew symmetry, $f(u,v) = -f(v,u)$ for all $(u,v) \in V^2$,
   or, $f(u,v) = -f(v,u)$ for all $(u,v) \in X^2$,
   or, $\sum_{u \in X} \sum_{v \in X} f(u,v) = \sum_{u \in X} \sum_{v \in X} (-f(v,u))$,
   or, $f(X,X) = -f(X,X)$. □

2. $\boxed{f(X,Y) = -f(Y,X) \text{ for } X,Y \subseteq V}$

   Proof: From skew symmetry, $f(u,v) = -f(v,u)$ for all $(u,v) \in V^2$,
   or, $f(u,v) = -f(v,u)$ for all $(u,v) \in X \times Y$,
   or, $\sum_{u \in X} \sum_{v \in Y} f(u,v) = \sum_{u \in X} \sum_{v \in Y} (-f(v,u))$,
   or, $\sum_{u \in X} f(u,Y) = \sum_{u \in X} (-f(Y,u))$, or, $f(X,Y) = -f(Y,X)$. □

3. $\boxed{\begin{array}{l} f(X \cup Y, Z) = f(X,Z) + f(Y,Z) \text{ and } f(Z, X \cup Y) = f(Z,X) + f(Z,Y) \\ \text{for } X,Y,Z \subseteq V, X \cap Y = \emptyset \end{array}}$

   Proof: $f(X \cup Y, Z) = \sum_{u \in X \cup Y} f(u,Z) = \sum_{u \in X} f(u,Z) + \sum_{u \in Y} f(u,Z)$ [since $X \cap Y = \emptyset$]
   $= f(X,Z) + f(Y,Z)$.
   The other proof is similar. □

4. $\boxed{|f| = f(V,t)}$

   Proof: $|f| = f(s,V) = f(V,V) - f(V \setminus \{s\}, V)$ [by previous property]
   $= f(V, V \setminus \{s\})$ [since $f(V,V) = 0$]
   $= f(V,t) + f(V, V \setminus \{s,t\})$ [by previous property]
   $= f(V,t)$ [since $f(V, V \setminus \{s,t\}) = 0$].

   How $f(V, V \setminus \{s,t\}) = 0$? By flow conservation, $f(u,V) = 0$ for every vertex $u \in V \setminus \{s,t\}$. Hence summing up those $f(u,v)$s for $u \in V \setminus \{s,t\}$, we get $f(V \setminus \{s,t\}, V) = 0$, or, $f(V, V \setminus \{s,t\}) = 0$ [by Property 2 of this lemma]. □

3.2 Ford-Fulkerson Algorithm

It starts with $f(u,v) = 0\ \forall (u,v) \in V^2$. Iteratively, it increases the flow by finding an ***augmenting path*** p from s to t along which more flow can be pushed. The flow along p is given by the minimum value of the capacities of the edges comprising p; and an/the edge having minimum capacity in p is called ***critical edge***.

Given an existing flow f in G, the maximum amount of additional flow that can still be pushed from any vertex $u \in V$ to any vertex $v \in V$ is defined as the ***residual capacity*** of (u,v), given by

$$c_f(u,v) = c(u,v) - f(u,v). \tag{3.2}$$

Example: Let $c(u,v) = 10, c(v,u) = 0$. Then $f(u,v) = 6$ yields $c_f(u,v) = 10 - 6 = 4$ and $c_f(v,u) = 0 - (-6) = 6$.

The ***residual network*** of G induced by f is $G_f = (V, E_f)$, where $E_f = \{(u,v) \in V^2 : c_f(u,v) > 0\}$. The edges in E_f are called ***residual edges***. We have the following lemma on the size of E_f.

Lemma 3.2.1 $|E_f| \leqslant 2|E|$.

Proof: $(u,v) \in E_f$ only if (i) $(u,v) \in E$ or (ii) $(u,v) \notin E$ but $(v,u) \in E$.
Hence the proof. □

From G_f, we can augment a larger flow in G, as stated in the following lemma.

Lemma 3.2.2 *If f is a flow in G and f' is a flow in G_f, then $f + f'$ is a flow in G whose value is $|f + f'| = |f| + |f'|$.*

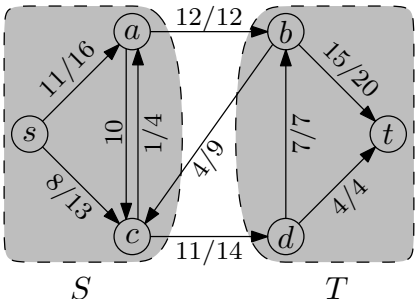
Proof: We first prove the three flow properties for $f + f'$ on G.
Skew symmetry: For all $(u,v) \in V^2$, $(f + f')(u,v) = f(u,v) + f'(u,v) = -f(v,u) - f'(v,u) = -(f + f')(v,u)$.
Capacity constraints: For all $(u,v) \in V^2$, $(f + f')(u,v) = f(u,v) + f'(u,v) \leqslant f(u,v) + c(u,v) - f(u,v)$ [since $f'(u,v) \leqslant c_f(u,v) = c(u,v) - f(u,v)$], or, $(f + f')(u,v) \leqslant c(v,u)$.
Flow conservation: For all $u \in (V \setminus \{s,t\})$, $(f + f')(u,V) = f(u,V) + f'(u,V) = 0 + 0 = 0$.

Now, $|f + f'| = (f + f')(s,V) = f(s,V) + f'(s,V) = |f| + |f'|$, which completes the proof. □

From Lemma 3.2.2, it's clear that if we get an ***augmenting path*** p in G_f, and find its ***residual capacity***, given by

$$c_f(p) = \min\{c_f(u,v) : (u,v) \in p\},$$

then $f_p = c_f(p)$ is a flow in G_f, which augments the flow in G from f to $f + f_p$. Iteratively, $f + f_p$ induces another residual network, and so on. When shall it converge to yield the max flow? It's obviously when the residual network contains no augmenting path. The ***Max-flow Min-cut Theorem*** answers this, which is explained next.



A ***cut***(S,T) of $G(V,E)$ is a partition of V into S and $T = V \setminus S$ such that $s \in S$ and $t \in T$. If f is a flow in G then the ***net flow*** across this cut is defined as $f(S,T)$, and its ***capacity*** is $c(S,T)$. In the figure aside, $f(S,T) = 12-4+11 = 19$, and its capacity is $c(S,T) = 12+14 = 26$.

Note: $f(S,T)$ consists of all flows from S to T, but $c(S,T)$ only the positive capacities.

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

Lemma 3.2.3 (Flow-cut lemma) *If f be any flow in G, then for any cut (S,T) of G, $f(S,T) = |f|$.*

Proof: Based on the properties 1 and 3 of Lemma 3.1.1.

$$
\begin{aligned}
f(S,T) &= f(S,V) - f(S,S) \text{ [By Lemma 3.1.1-3, } f(S, V\setminus S) = f(S,V) - f(S,S)] \\
&= f(S,V) \text{ [By Lemma 3.1.1-1, } f(S,S) = 0] \\
&= f(s,V) + f(S\setminus\{s\},V) \text{ [By Lemma 3.1.1-3]} \\
&= f(s,V) \quad \left|\begin{array}{l} \text{By Lemma 3.1.1-3. Since } t \notin S, \text{ we get } S\setminus\{s,t\} = S\setminus\{s\}. \\ \text{Thus, by flow conservation, } f(u,V) = 0\ \forall u \in S\setminus\{s\}, \text{ which sums to} \\ f(S\setminus\{s\},V) = 0. \end{array}\right. \\
&= |f|.
\end{aligned}
$$

□

From Flow-cut lemma (Lemma 3.2.3), it's evident that $\boxed{|f| \leqslant c(S,T) \text{ for } \textit{any} \text{ cut } S(,T)}$. This gives Max-flow Min-cut Theorem, stated as follows.

Theorem 3.2.4 (Max-flow Min-cut Theorem) *If f is a flow in G, then the following conditions are equivalent.*

1. *f is a max flow in G.*
2. *G_f contains no augmenting path.*
3. *$|f| = c(S,T)$ for a cut (S,T) having minimum capacity (min-cut).*

Proof: (1) ⇒ (2) : By contradiction. If G_f has some augmenting path p, then some more flow f' can be shipped through p, resulting to increase in flow from f to f'.

(2) ⇒ (3) : Define a cut as follows.
$S = \{u \in V :$ there exists a path from s to u in $G_f\}$, and $T = V \setminus S$.
Now, for each $(u \in S, v \in T)$, we have $f(u,v) = c(u,v)$; for, otherwise $(u,v) \in E_f$, which implies $v \in S$—a contradiction. Hence, by Flow-cut lemma (Lemma 3.2.3), $|f| = f(S,T) = c(S,T)$, which is a min-cut as its capacity is minimum and all other cuts of G have $c \geqslant |f|$.

(3) ⇒ (1) : Since $|f| \leqslant c(S,T)$ for any cut (S,T) as explained earlier in consequence of Flow-cut lemma (Lemma 3.2.3), the condition $f = c(S,T)$ implies that f is a max flow. □

Algorithm 11: FORD-FULKERSON(G, s, t).

```
for each (u, v) ∈ E do
    f(u, v) ← f(v, u) ← 0
while ∃ p(s ⇝ t) ∈ G_f do
    c_f(p) ← min{c_f(u, v) : (u, v) ∈ p}
    for each (u, v) ∈ p do
        f(u, v) ← f(u, v) + c_f(p)
        f(v, u) ← −f(u, v)
```

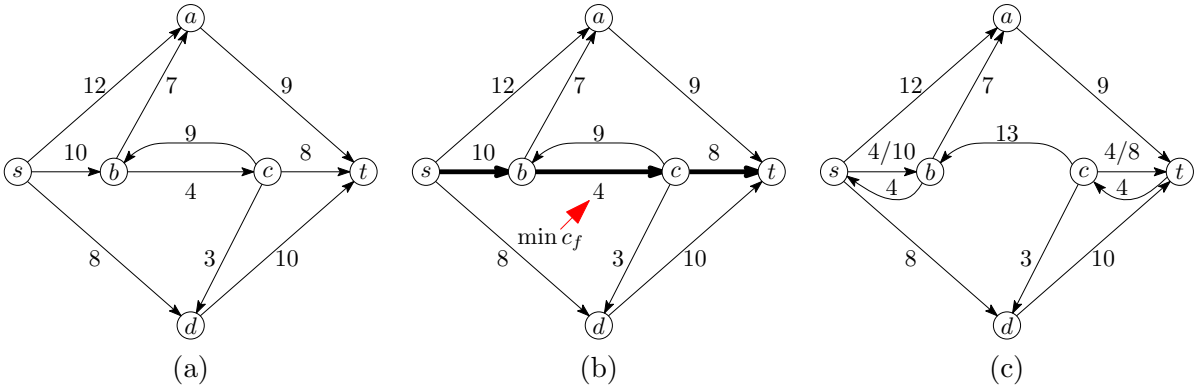


Figure 3.1: (a) A flow network G with its edge capacities. (b) An augmenting path $p = \langle s, b, c, t\rangle$ with $c(p) = \min\{c(s, b), c(b, c), c(c, t)\} = 4$. (c) The residual network G_f after augmenting a flow of value $|f| = 4$ through p. Note how $c(b, s)$ changes 0 to 4, as $c(b, s) = 0 - (-|f|) = 4$ after the augmentation. Similar is the case of $c(t, c)$. The residual capacity of (c, b) also gets changed to $9 - (-4) = 13$.

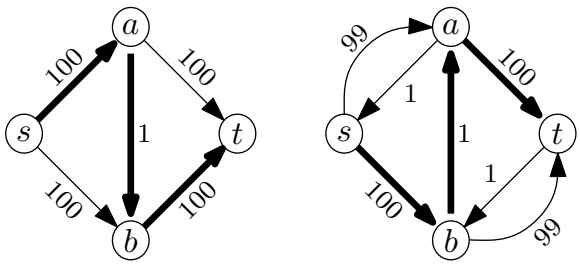


Figure 3.2: Two iterations of the basic Ford-Fulkerson algorithm. In each iteration, the augmenting path (shown in bold) is not the shortest path ((s, a, t) or (s, b, t)) from s to t in G or G_f with unit-weight edges and the flow capacity of the augmenting path is 1. Had the augmenting paths been (s, a, t) and (s, b, t) in the first two iterations, we could have obtained the max-flow $|f| = 100 + 100 = 200$ in just two iterations! In fact, Edmonds-Karp version gets it in two iterations by using the two shortest paths.

The basic Ford-Fulkerson algorithm: It applies DFS first on G and then iteratively on G_f with s as the start vertex to find an augmenting path p (from s to t); see Algo 11 and Fig. 3.1. If all the edges of G have integer (or 'integer-able') capacities—as mostly found in practice, then Algo 11 may iterate as many as $|f|$ times, thus consuming $O(E|f|)$ time in the

worst case ($O(E)$ time for Steps 1–2 and $O(E) \times |f|$ for the **while** loop (Steps 3–7)). A typical example is given in Fig. 3.2. As $|f|$ may be unusually large, the algorithm is not ready to get explained in terms of an efficient time complexity.

3.2.1 Edmonds-Karp algorithm[1]

It is irrespective of integer or non-integer capacities. It finds the shortest path from s to t in G_f, considering all its edges with unit weight (or by BFS on unweighted G_f). To prove its $O(VE^2)$ time complexity, we need the following lemma.

Lemma 3.2.5 *For each vertex $v \in V \setminus \{s,t\}$, the shortest-path distance $\delta_f(s,v)$ of v from s never decreases with successive flow augmentations.*

Proof: For contradiction, let $V' = \{v' : \delta_{f'}(s,v') < \delta_f(s,v')\}$ when a flow f is augmented to f'. Let $v \in V'$ be such that $\delta_{f'}(s,v) = \min\{\delta_{f'}(s,v') : v' \in V'\}$. Then, $\delta_{f'}(s,x) < \delta_{f'}(s,v)$ implies that $x \notin V'$, i.e.,

$$\delta_{f'}(s,x) \geqslant \delta_f(s,x). \tag{3.3}$$

Let $p' := s \rightsquigarrow u \to v \rightsquigarrow t$ be the shortest path in $G_{f'}$. Then

$$\begin{aligned} & \delta_{f'}(s,u) = \delta_{f'}(s,v) - 1 \\ \text{or,} \quad & \delta_{f'}(s,u) < \delta_{f'}(s,v) \end{aligned} \tag{3.4}$$

which, from Eq. 3.3, implies that

$$\delta_{f'}(s,u) \geqslant \delta_f(s,u). \tag{3.5}$$

Now consider the following two possible cases.

- Case $f(u,v) < c(u,v)$:
  Here (u,v) is not a critical edge and so may or may not lie on the shortest path from s to v. Hence, $\delta_f(s,v) \leqslant \delta_f(s,u) + 1$, where the strict inequality corresponds to the case when the shortest path from s to v does not contain u and is shorter than any path via u. So, from Eq. 3.5, $\delta_f(s,v) \leqslant \delta_{f'}(s,u) + 1 = \delta_{f'}(s,v)$ by Eq. 3.4, which is a contradiction.
- Case $f(u,v) = c(u,v)$:
  Here (u,v) is a critical edge, and so $(u,v) \notin E_f$. But, $(u,v) \in E_{f'}$ by original supposition. Hence, there should be a flow along (v,u) in the augmenting path chosen in G_f, and the augmenting path is $s \rightsquigarrow v \to u \rightsquigarrow t$. So,

  $$\begin{aligned} \delta_f(s,v) &= \delta_f(s,u) - 1 \\ &\leqslant \delta_{f'}(s,u) - 1 \quad \text{(from Eq. 3.5)} \\ &= \delta_{f'}(s,v) - 2 \quad \text{(from Eq. 3.4)} \end{aligned}$$

  or, $\delta_{f'}(s,v) > \delta_f(s,v)$ — a contradiction again.

[1] Jack Edmonds and Richard M. Karp (1972). Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM* **19**(2): 248–264.

□

Time complexity: Lemma 3.2.5 can be used to prove that Edmonds-Karp algorithm has $O(VE^2)$ time complexity. The proof is as follows.

Let (u, v) be *critical* for the first time when the flow is f. Then

$$\delta_f(s, v) = \delta_f(s, u) + 1 \tag{3.6}$$

and $(u, v) \notin E_f$.

The edge (u, v) can reappear later in an augmenting path only if (v, u) occurs on some augmenting path before that, say, for flow f'. And then,

$$\begin{aligned} \delta_{f'}(s, u) &= \delta_{f'}(s, v) + 1 \\ &\geqslant \delta_f(s, v) + 1 \quad \text{(from Lemma 3.2.5)} \\ &= \delta_f(s, u) + 2 \quad \text{(from Eq. 3.6).} \end{aligned}$$

Thus, for each successive time (u, v) becomes critical, its δ from s increases by at least 2. And its value ranges from 1 to $V - 2$. So, (u, v) can become critical at most $O(V)$ times. This implies that the edges of G become critical at most $O(VE)$ times in total. An augmenting path contains one or more critical edges and can be found in $O(E)$ time by DFS. Hence, the total time complexity is $O(VE^2)$.

3.3 Maximum Bipartite Matching

Matching: A ***matching*** (or ***independent edge set***) in an undirected graph $G(V, E)$ is a subset of edges $M \subseteq E$ in which no two edges share a common vertex. A vertex $v \in V$ is said to be ***matched*** by M if some edge of M is incident on v; otherwise, v is ***unmatched***.

M is said to be a ***maximal matching*** if no more edge can be added from $E \setminus M$ to M; that is, M is maximal if it is not a proper subset of any other matching in G (Fig. 3.3).

If the matching M has maximum cardinality, then it is called a ***maximum matching*** (Fig. 3.3). Note that every maximum matching is always maximal, but not the reverse. A ***perfect matching*** (1-factor matching) is a matching which matches all vertices of G. that is, every vertex of the graph is incident to exactly one edge of the matching.

Figure 3.3: Maximal matching (left) versus maximum matching (right). The maximum matching here is also a perfect matching. Edges in the matching are highlighted in gray.

Maximum bipartite matching: A ***bipartite graph*** is an undirected graph $G(V, E)$ in which V can be partitioned into two sets, X and Y, such that each edge of E has one vertex in X and the other vertex in Y. Since a matching M in any graph $G(V, E)$ is a subset of E, each edge of M for a bipartite matching naturally has one vertex in X and the other vertex in Y.

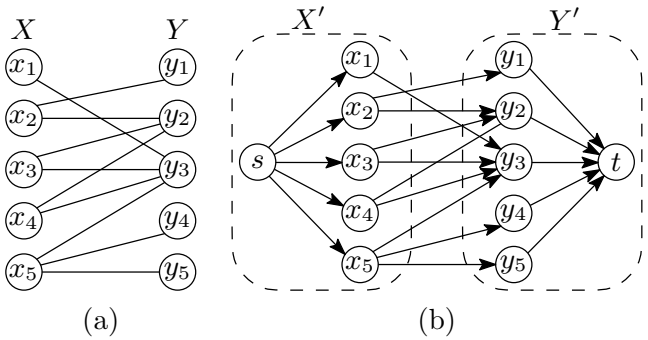


Figure 3.4: (a) A bipartite graph $G(V, E)$ with $V = X \cup Y$ as the bipartite set of vertices. (b) The flow network G' constructed from G.

We can use the Ford-Fulkerson algorithm to find a maximum matching in a bipartite graph $G(V, E)$ with the partition $V = X \cup Y$. For this, we first construct the flow network $G' = (V', E')$ from G as follows. Take a source s and a sink t, and define $X' = X \cup \{s\}$, $Y' = Y \cup \{t\}$ so that $V' = V \cup \{s, t\}$; define $E' = \{(s, u) : u \in X\} \cup \{(u, v) : u \in X, v \in Y, (u, v) \in E\} \cup \{(v, t) : v \in Y\}$ (all directed edges). Assign unit capacity to each edge in E'. The max-flow in G' will be a positive integer that equals the cardinality of maximum matching in G. It's based on the Lemma 3.3.1 and Integrality theorem (Theorem 3.3.2).

Lemma 3.3.1 *M is a matching in G if and only if there is a flow f in G' such that $|f| = |M|$.*

Proof: To show that a matching (whether max or not) M in G corresponds to a flow f in G', define f as follows. For each $(u, v) \in M$, assign $f(s, u) = f(u, v) = f(v, t) = 1$ and $f(u, s) = f(v, u) = f(t, v) = -1$. For each $(u, v) \in E' \setminus M$, assign $f(u, v) = 0$. The flow properties (capacity constraint, skew symmetry, and flow conservation) can be easily verified for each such unit-value flow from s to t containing $(u, v) \in M$. The net flow across the cut (X', Y') is equal to $|M|$, and so $|M| = |f|$ by flow-cut lemma (Lemma 3.2.3).

Conversely, if f is an integer-valued flow in G', then define $M = \{(u, v) : u \in X, v \in Y, f(u, v) = 1\}$. Since each $u \in X$ has one entering edge (s, u) with $c(s, u) = 1$, by Integrality theorem, $f(s, u) \in \{0, 1\}$. Hence, by flow conservation, $f(u, Y) \in \{0, 1\}$, which implies that at most one edge leaving from u is included in M. A symmetric argument shows that at most one incoming edge to each $v \in Y$ has a 1-unit flow and included in M. Thus, M is a valid matching in G. To show $|M| = |f|$, we use the just-proven fact that $f(u, v) = 1$ iff $(u, v) \in M$. Hence,

$$
\begin{aligned}
|M| &= f(X, Y) \\
&= f(X, V') - f(X, s) - f(X, X) - f(X, t) \\
&= 0 + f(s, X) - 0 - 0 \quad \left| \begin{array}{l} \text{By flow conservation, } f(X, V') = 0; \\ \text{by skew symmetry, } f(X, s) = -f(s, X); \\ \text{by Subgraph Flow (Lemma 3.1.1): Property 1, } f(X, X) = 0; \\ f(X, t) = 0, \text{ as there is no edge from } X \text{ to } t. \end{array} \right. \\
&= f(s, V') \quad [\text{Since there is no edge from } s \text{ to } V' \setminus X] \\
&= |f|.
\end{aligned}
$$

□

Result: As $M \Leftrightarrow f$ with $|M| = |f|$, we get $\max\{|M|\} \Leftrightarrow \max\{|f|\}$.

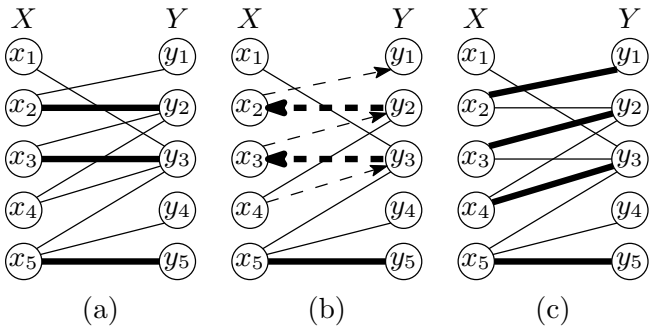


Figure 3.5: (a) A bipartite graph with a matching M (bold edges). (b) The augmenting path (dashed) in the residual graph. (c) The maximum matching (bold edges) obtained by the augmentation.

Theorem 3.3.2 (Integrality theorem) *If the edge capacities of a flow network are integer-valued, then by Ford-Fulkerson algorithm, the flows across all edges, and hence the max flow f, are all integer-valued.*

Proof: By induction on the number of iterations. □

Alternating nature of augmenting paths: If a matching M is not maximum in G, then the corresponding flow f in G' is also not maximum, and so there is an augmenting path p in the residual network G'_f (Fig. 3.5). Some edges in this path p are backward (i.e., directed from Y to X) and some forward (directed from X to Y), although initially all edges in G' were directed from X to Y! Augmenting paths therefore alternate between edges used backward and forward, and hence also called ***alternating paths*** in the context of graph matching. The effect of this augmentation is to take the backward edges out of the matching and replace them with the forward edges. Because the augmenting path goes from s to t, there is always one more forward edge than backward edge, thereby increasing the matching size by one.

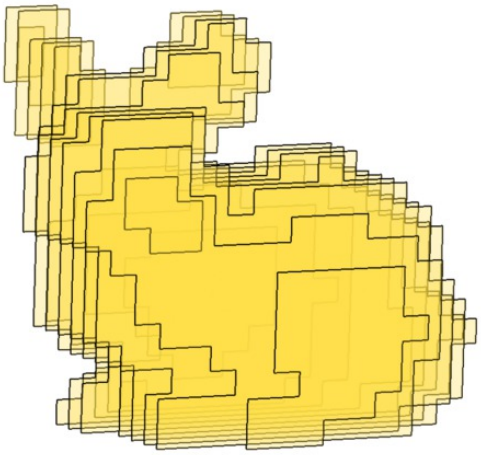
Books

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (1990). *Introduction to Algorithms*. MIT Press (Indian Reprint by PHI).

2. Jon Kleinberg and Éva Tardos (2006). Algorithm Design. Pearson Education (India).

Chapter 4

Plane Sweep Algorithms



A sweep line or plane sweep technique is an important technique used to solve many problems in computational geometry. It uses a conceptual sweep line (or sweep surface) that moves across the plane (space), stopping at some event points. Geometric operations are restricted to geometric objects that either intersect or are in the vicinity of the sweep line at a particular instant, and the complete solution is available once the line has passed through all the objects or the event points.

The figure aside shows a set of slices obtained by sweeping the 3D space containing an orthogonal polyhedron that encapsulates a real-world object. The algorithm is developed by us in 2012 and presented in *15th International Workshop on Combinatorial Image Analysis (IWCIA 2012)*, Austin, Texas, USA. For more details, visit: http://link.springer.com/chapter/10.1007/978-3-642-34732-0_2

4.1 Line Segment Intersection

Given a set $S = \{s_1, \ldots, s_n\}$ of line segments, the problem is to find their points of intersection. We should avoid testing pairs of segments lying far apart to lower the computation time from $O(n^2)$ to something asymptotically less. For this, we have an easy observation: If the y-intervals of two segments do not overlap, then they would never intersect. But finding such segments in pairs again takes $O(n^2)$ time, which is same as exhaustive testing. Instead, we can sort the segments in descending order of their y-coordinates, and then consider a ***sweep line*** λ moving vertically downwards to find the intersection among all the segments intersected by λ at "appropriate instants"—a typical ***plane sweep algorithm***. The issues are:

1. How to determine the "appropriate instants"? We will see it shortly; more specifically, they correspond to *event points*.
2. At every or many event points, there may be $O(n)$ segments intersected by λ. Then should we check intersections among all these segments taken in pairs? No way! It'll again give us $O(n^2)$ time for a single instance of the sweep line.

Hence, we should verify the intersection of two segments only when they are "close enough". This is done by using the left-to-right ordering of the segments intersected by λ. If two segments are not adjacent to one another along λ, then we need not verify. It is based on the fact that if p is the point of intersection between two segments, and no other segment passes through or is incident at p, then these two segments would be adjacent along λ for a certain stretch of time while λ is moving down (e.g., (s_2, s_1) and (s_2, s_4) in Fig. 4.1). Interestingly, we would detect that at the right instant or right event point!

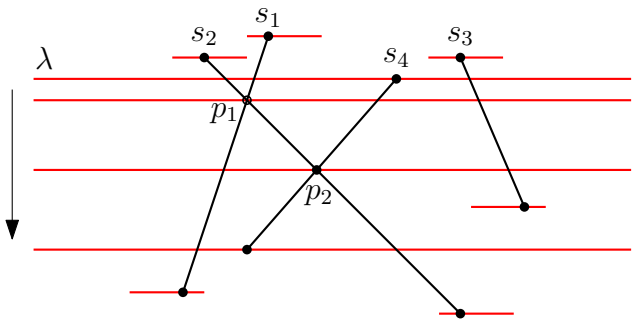


Figure 4.1: Illustration on how adjacent segments along the sweep line λ (shown in red) go on changing at event points as λ moves downward. Some positions of λ are shown short for clarity. Two segments are said to be "adjacent" if their points of intersection with λ are adjacent in the sequence of points of intersection of all the segments with λ.

4.1.1 Event points

An event point is such a point where the status of λ changes. Clearly, the status of λ is defined by the *ordered set* of segments intersecting λ. All event points are stored in the ***event queue***, Q, and processed one by one when encountered by λ. An event point p_2 can be one of the following:

Upper endpoint of a segment s; for a horizontal segment, it's the left endpoint.
Example: Fig. 4.1, when λ reaches the upper endpoint of s_4:
s_4 appears as the new segment intersecting λ, and so the new sequence of segments intersecting λ becomes $\langle s_2, s_1, s_4, s_3 \rangle$.
Actions taken: s_4 is deleted from Q and inserted in the data structure T storing the status of λ. Since s_4 is new in λ, its left adjacent segment becomes s_1, its right adjacent segment becomes s_3, and so s_1 and s_3 do not remain adjacent. Possible intersections for the newly formed adjacent pairs, (s_1, s_4) and (s_4, s_3), are verified; none is found, resulting to no insertion in Q.

Lower endpoint of a segment s; for a horizontal segment, it's the right endpoint.
Example: Fig. 4.1, when λ reaches the lower endpoint of s_4:
s_4 disappears from the sequence; the new sequence of segments intersecting the sweep line becomes $\langle s_1, s_2 \rangle$.
Actions: s_4 is deleted from T. Newly formed adjacent pair is (s_1, s_2), whose intersection point p_1 is verified and found to exist above λ. This means p_1 has already been inserted in and deleted from Q, after necessary processing.

Point of intersection between two segments. It's not known beforehand, but detected on the fly while λ moves down and finds each intersection point corresponding to its two adjacent segments for the first time.
Example: Fig. 4.1: p_2 is detected when s_2 and s_4 become adjacent at p_1. The point p_1, in turn, was computed and inserted in Q earlier when s_2 and s_1 became adjacent for the first time while the sweep line was passing through s_2.
As λ passes through p_2, the sequence of intersecting segments changes from $\langle s_1, s_2, s_4, s_3 \rangle$ to $\langle s_1, s_4, s_2, s_3 \rangle$, causing just a swap in the positions of s_2 and s_4 in the sequence. Hence, only the adjacent segments of s_2 and s_4 would change, for which we have to verify at most two points of intersection: one between s_4 and its left neighbor (s_1), and another between s_2 and its right neighbor (s_3).

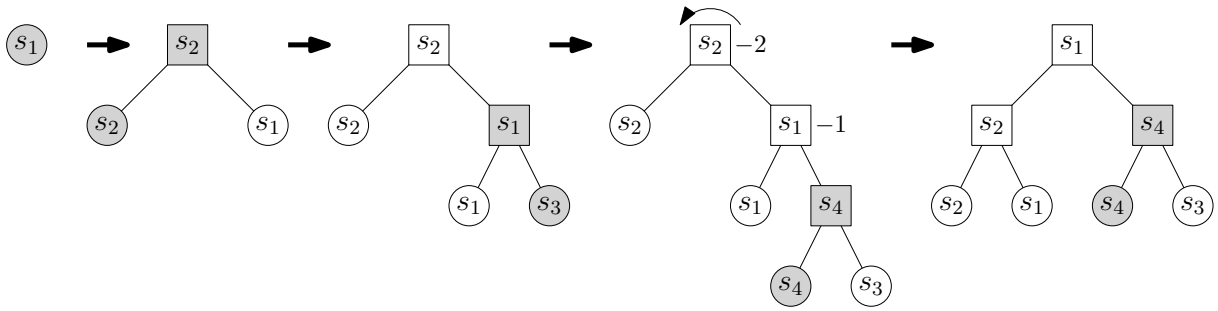


Figure 4.2: Illustration on how T goes on changing as λ moves downward and new event points are encountered in Fig. 4.1. As λ encounters the upper endpoint of s_4, the segment s_4 is inserted in T as a leaf node together with the required non-leaf node containing s_4 (shown in gray). As T becomes unbalanced after this insertion, left rotation is applied at the root node containing s_2 (balance factor $= -2$) to restore balance in T.

Results:

1. As each intersection point is obtained, it is inserted (not typically 'enqueued', as explained later) in Q.
2. Given that each event point p_2 is either an upper/lower endpoint of a unique segment or the point of intersection of two segments, we need to verify at most two intersection points arising out of two newly formed pairs of adjacent segments in the sequence of λ-intersecting segments.

4.1.2 Data Structures

The *event queue* Q initially contains the upper and the lower endpoints of all the given segments. Further, all intersection points, as detected on the fly, have to be inserted in the event queue Q. As the sweep line λ moves down, the events are handled by lexicographically decreasing order of their (y, x)-coordinates. Hence, Q should be a priority queue. But as there might be an event point already in Q, and that event point may be some upper endpoint or lower endpoint or even an intersection point, a newly detected intersection point p cannot be simply inserted in Q. We have to search whether the intersection point p (if not lying above λ) is already there in Q. For this, a max-heap is not an efficient realization of Q. Instead, a height-balanced search tree can be used. Then each operation—whether searching for an existing event point or insertion of a new event point or fetching the next event point—can be performed in a time logarithmic in the size of Q, or $O(\log n)$, where n is the number of segments, since Q is $O(n)$ in size, as explained in Sec. 4.1.4.

To maintain the status of λ through T, we need to insert newly intersecting segments or delete existing segments from T. We have to also know the adjacent segments from T whenever there is a change in the status of λ. All these can be done efficiently if T is also a height-balanced search tree. The leaf nodes of T will actually contain all the segments currently intersecting λ; and each non-leaf node of T will contain the segment of the rightmost node of its left subtree (Fig. 4.2). This enables a simpler realization of searching a segment in T using the segments stored in the non-leaf nodes. For example, if we want to find the left adjacent segment of a

Algorithm 12: FINDINTERSECTIONS(S)

```
1  Insert the endpoints of all segments of S in Q.
2  Initialize T as empty.
3  while Q ≠ { } do
4  |  Delete the point p with max y from Q
5  |  Rebalance Q if needed
6  |  HandleEventPoint(p)
```

Procedure HandleEventPoint(p). $left(p)$ and $right(p)$ denote the respective nodes that lie left and right of p in T; LeftMost($U_p \cup I_p$) and RightMost($U_p \cup I_p$) denote the leftmost the rightmost segments in $U_p \cup I_p$.

```
1  U_p ← segments in Q whose upper endpoint is p
2  L_p ← segments in T whose lower endpoint is p
3  I_p ← segments in T whose interior point is p
4  if |U_p ∪ L_p ∪ I_p| > 1 then
5  |  Report p, together with U_p, L_p, I_p
6  Delete L_p ∪ I_p from T
7  Insert U_p ∪ I_p in T
8  if U_p ∪ I_p = ∅ then
9  |  FindNewEvents(left(p), right(p), p)
10 else
11 |  FindNewEvents(left(LeftMost(U_p ∪ I_p)), LeftMost(U_p ∪ I_p), p)
12 |  FindNewEvents(right(RightMost(U_p ∪ I_p)), RightMost(U_p ∪ I_p), p)
```

Procedure FindNewEvents(s, s', p)

```
1  if s and s' intersect at p' so that p' lies below p or horizontally to its right and p' is not in
   Q then
2  |  Insert p' in Q
```

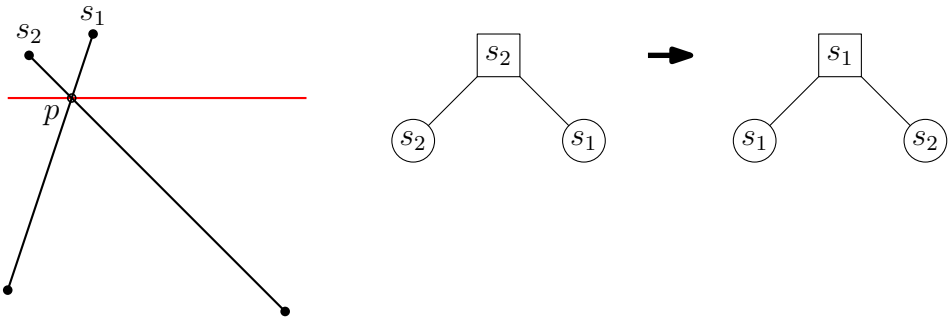


Figure 4.3: A small-yet-representative illustration on how deletion of $I_p = \{s_2, s_1\}$ from T and then insertion of the same (i.e., *same ordered set*) $I_p = \{s_2, s_1\}$ in T reverses the left-to-right ordering of the segments from I_p correctly!

point p on λ, we can use the segments of the non-leaf nodes to binary-search the leaf node that contains the required segment.

4.1.3 Algorithm

See Algorithm 12 and the two related procedures. It's by Bentley and Ottmann[1]. Left-to-right ordering of segments along λ should have left-to-right ordering of their leaf nodes in T. In Step 7 of HandleEventPoint, this is ensured. A node ν deleted before another, ν', when inserted after ν', yields the reverse ordering of ν and ν' in such a tree. See the illustration in Fig. 4.3. In FindNewEvents, on checking in Q, if p' is found, then p' was surely inserted earlier in Q as an upper or a lower endpoint, or as an intersection point. Whatever be the case, we need not re-insert p'—even if p' is an intersection point of some other segment pair, since all these segments would be rightly fetched from Q and T when λ reaches p' (as shown in Steps 1–3 of HandleEventPoint).

4.1.4 Time complexity

Insertion with assurance to appropriate ordering of upper and lower endpoints of n segments in Q (height-balanced search tree) takes $O(n \log n)$ time. For each point of intersection p, a new event point is searched in Q and inserted by FindNewEvents, if required. If v be the sum of $v_p = |U_p \cup L_p \cup I_p|$ over all event points, then for the planar graph (connected or disconnected) G induced by S, we get $v \leqslant 2n + I = O(n + I)$, where I is the number of points of intersection in S. As $e = O(v)$ for a planar graph with v vertices and e edges, and each insertion/deletion in/from Q or T takes $O(\log n)$ time, the total time is $O((n + I) \log n)$.

However, the first-principle proof follows from the following facts:

- $v \leqslant 2e$, since each vertex is incident to at least one edge.
- $3f \leqslant 2e$, since an edge is incident on at most two faces and a face has at least three edges.
- $v - e + f \geqslant 2$ by Euler's formula for any (disconnected or connected) planar graph G, where v, e, f denote the respective number of vertices, edges, and faces in G.

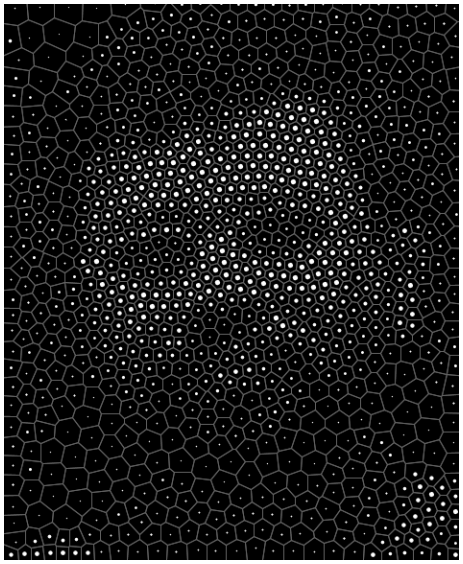
Hence, $2n + I - e + 2e/3 \geqslant 2$, or, $e \leqslant 6n + 3I - 6$ and $v \leqslant 12n + 6I - 12$. Thus, both e and v depend linearly on $n + I$.

Books

1. M. D. Berg, M. V. Kreveld, M. Overmars, and O. Schwarzkopf (2000). *Computational Geometry Algorithms and Applications*, Springer-Verlag, Berlin.

[1] J. L. Bentley, and T. A. Ottmann (1979), Algorithms for reporting and counting geometric intersections, *IEEE Transactions on Computers*: C-28(9), pp. 643–647, doi:10.1109/TC.1979.1675432.

4.2 Voronoi Diagram



Historically, the use of ***Voronoi diagrams*** can be traced back to Descartes in 1644. A Voronoi diagram is also called ***Voronoi tessellation*** or ***Voronoi decomposition*** (named after Russian mathematician Georgy Fedoseevich Voronoi who defined and studied the general d-dimensional case in 1908), or ***Dirichlet tessellation*** (after Lejeune Dirichlet). Dirichlet used 2D and 3D Voronoi diagrams in his study of quadratic forms in 1850. Voronoi diagrams are used to solve various physical and biological problems. For example, British physician John Snow used a Voronoi diagram in 1854 to illustrate how the majority of people who died in the Soho cholera epidemic lived closer to the infected Broad Street pump than to any other water pump.

The figure aside shows an artistic rendition on the image of my face. I have created it simply inspired by Voronoi diagram. For more details, visit:
http://www.evilmadscientist.com/2012/stipplegen-weighted-voronoi-stippling-and-tsp-paths-in-processing/

In general, a ***Voronoi diagram*** is a special kind of decomposition of a metric space determined by distances to a specified discrete set of objects in the space, e.g., a set of points called ***sites***. Here we consider the Euclidean metric space to find out the Voronoi diagram of a set of sites in 2D. The set of all points closer to a site p of P than to any other site of P is the interior of a (in some cases unbounded) convex polygon called the ***Voronoi cell*** (or ***Dirichlet domain***) for p. The set of such convex polygons tessellates the whole space, and is the ***Voronoi tessellation*** corresponding to the set P.

We start with some definitions related with Voronoi diagram. A point $p_i = (x_i, y_i) \in \mathbb{R}^2$ that defines a Voronoi cell is termed as a ***site***, and the set $P = \{p_1, p_2, \ldots, p_n\}$ is called the ***set of sites***. The Euclidean distance, denoted by $d(p, q) = \sqrt{(x_p - x_q)^2 + (y_p - y_q)^2}$, is considered here as the ***distance*** between two points or sites, $p(x_p, y_p)$ and $q(x_q, y_q)$ in $\mathbb{R}^2$. The straight line segment joining two points or sites, p and q in $\mathbb{R}^2$, is termed as the ***line segment*** $\overline{pq}$. If p and q be any two points in $\mathbb{R}^2$, then the xy-plane is divided into two ***half-planes*** by their perpendicular bisector, $b(p, q)$. The plane bounded by the and containing the point p is called the half-plane containing p, denoted by $h(p, q)$, and that bounded by $b(p, q)$ and containing q is called the half-plane containing q, denoted by $h(q, p)$. Observe that $h(p, q) \cup h(q, p) = \mathbb{R}^2$ for any $p, q \in \mathbb{R}^2$. Based on the above definitions, the Voronoi diagram is defined as follows.

Definition 4.2.1 (Voronoi diagram) *If $P = \{p_1, p_2, \ldots, p_n\}$ be the set of sites, then the Voronoi diagram of P, denoted by $VD(P)$, is the division of $\mathbb{R}^2$ into n regions, namely $vc(p_1)$, $vc(p_2), \ldots, vc(p_n)$, such that*

*(i) each region $vc(p_i)$ (called **Voronoi cell**) contains exactly one site, i.e., p_i;*

(ii) for any point q in each region $vc(p_i)$, p_i is the nearest site;
i.e., $d(q, p_i) = \min_{1 \leqslant j \leqslant n} d(q, p_j)$ for any $q \in vc(p_i)$, for each $p_i \in P$.

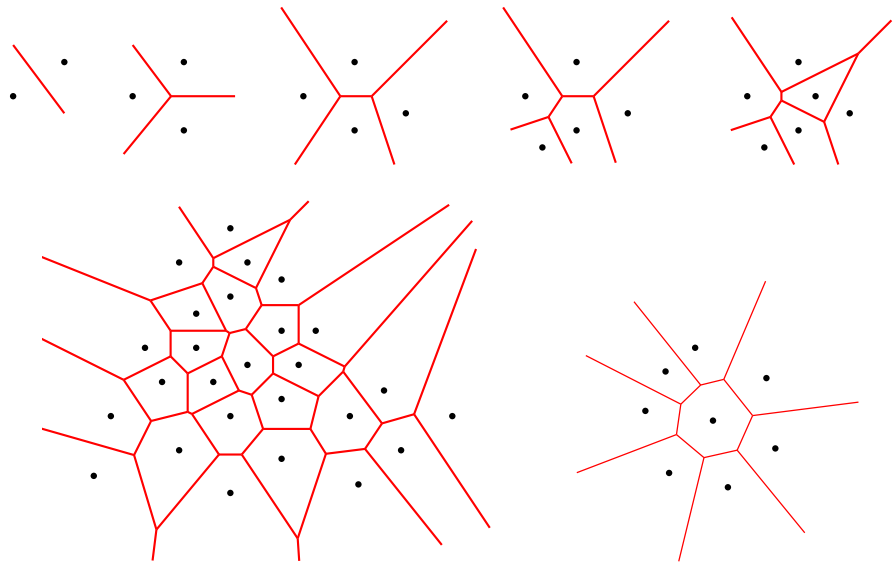


Figure 4.4: **Top:** Voronoi diagrams for 2, 3, 4, 5, and 6 sites. **Bottom-left:** Voronoi diagram for many sites. **Bottom-right:** A Voronoi diagram in which a Voronoi cell has $n-1$ vertices.

From Def. 4.2.1 (also see Fig. 4.4), it is evident that, for any point q in a Voronoi cell $vc(p_i)$, p_i is the nearer than any other site p_j, which means that q lies the half-plane $h(p_i, p_j)$ for any $p_j \neq p_i$. Hence, $vc(p_i) \subseteq h(p_i, p_j), \forall p_j \in P \setminus \{p_i\}$, which gives the following theorem.

Theorem 4.2.1 $vc(p_i) = \bigcap_{p_j \in P \setminus \{p_i\}} h(p_i, p_j)$.

Note: The dual graph[1] for a Voronoi diagram corresponds to the ***Delaunay triangulation*** for the same set of sites, P.

Note: The closest pair of sites corresponds to two adjacent Voronoi cells in $VD(P)$.

Note: The Voronoi diagram of a set of sites in d-dimensional space corresponds to the convex hull of the projection of the sites onto a $(d+1)$-dimensional paraboloid.

4.2.1 Vertices and Edges of a Voronoi Diagram

As shown in Fig. 4.4, a Voronoi cell can have $n-1$ vertices for a set of n sites. Hence, if we go by such straightforward analysis, then the total number of vertices of a Voronoi diagram can be as high as $n(n-1) = O(n^2)$.

However, the number of vertices and that of edges for any Voronoi diagram can be shown to be linear if we use some properties of a connected planar embedded graph[2], to which a Voronoi diagram can be augmented by introducing an extra vertex, say v_∞, and connecting all half-infinite edges to v_∞, as shown in Fig. 4.5.

[1] The ***dual graph*** of a graph $G = (V, E)$ is given by $G^* = (V^*, E^*)$ such that: (i) each vertex in V^* corresponds to a face of G; (ii) each face of G^* corresponds to a vertex of G; and (iii) two vertices in G^* are connected by an edge if the corresponding faces in G share an edge of G.

[2] A graph is ***planar*** if it can be drawn in a plane without graph edges crossing.

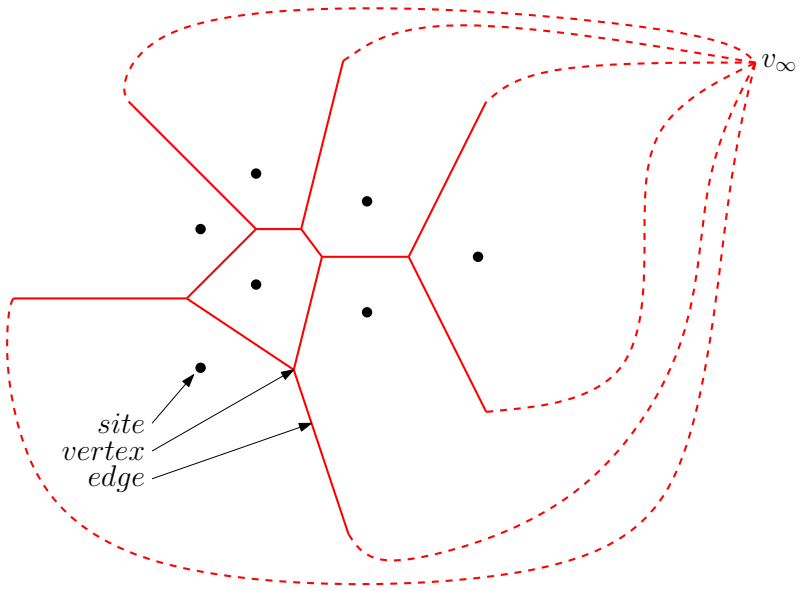


Figure 4.5: The Voronoi diagram (solid edges) and its augmented planar graph (dashed).

By Euler's formula, if m_v, m_e, and m_f be the number of vertices, number of edges, and number of faces of a connected planar embedded graph, then $m_v - m_e + m_f = 2$. Hence, if n_v, n_e, and n_f be the number of vertices, number of edges, and number of faces of $VD(P)$, then using the fact that $n_f = n$, we get:

$$(n_v + 1) - n_e + n = 2. \tag{4.1}$$

Now, at each vertex of $VD(P)$, at least three edges are incident. Hence, the sum of the degrees of all vertices of $VD(P)$ is at least $3(n_v + 1)$, from which we get:

$$2n_e \geqslant 3(n_v + 1). \tag{4.2}$$

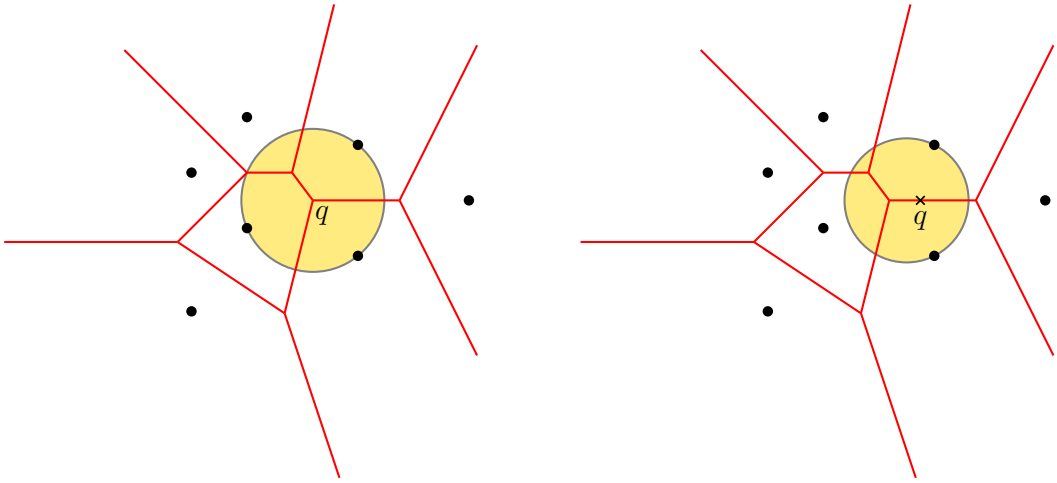
Combining Eqn. 4.1 and Eqn. 4.2, we get the maximum number of vertices and that of edges of $VD(P)$, as stated in the following theorem.

Theorem 4.2.2 *For $n \geqslant 3$, $VD(P)$ has $n_v \leqslant 2n - 5$ and $n_e \leqslant 3n - 6$.*

Theorem 4.2.2 indicates that, out of $\binom{n}{2}$ perpendicular bisectors for n sites in P, not all but some bisectors define the edges of $VD(P)$. Hence, the question that concerns us now is how to find the vertices and edges of $VD(P)$. To answer this, we use Def. 4.2.2, which helps in characterizing the vertices and the edges of $VD(P)$ as stated in Theorem 4.2.3.

Definition 4.2.2 *The largest empty circle centered at q w.r.t. P, denoted by $C_P(q)$, is the largest circle with center q and containing no site of P in its interior.*

Theorem 4.2.3 *(i) A point q is a vertex of $VD(P)$ if and only if $C_P(q)$ contains three or more sites on its boundary.*
(ii) The perpendicular bisector $b(p_i, p_j)$ defines (the interior of) an edge of $VD(P)$ if and only if there exists a point q on $b(p_i, p_j)$ such that $C_P(q)$ contains both p_i and p_j on its boundary but no other site.



The largest empty circle $C_P(q)$ with 3 or more sites on its boundary makes q a vertex of $VD(P)$.

The largest empty circle $C_P(q)$ with exactly 2 sites on its boundary makes q a point on an edge of $VD(P)$.

Figure 4.6: Largest empty circles centered about vertices and edge points of a Voronoi diagram.

Proof: (i) $\rightarrow$ For a point q, let $C_P(q)$ contains $k(\geqslant 3)$ sites on its boundary (Fig. 4.6). Then each of these k sites is nearest to q. Hence, q lies in the Voronoi cell of each of these k sites but does no lie in any other Voronoi cell of $VD(P)$, and q becomes a vertex of $VD(P)$.

$\leftarrow$ If q is a vertex of $VD(P)$, then at least $k(\geqslant 3)$ edges are incident at q, whereby q lies in k Voronoi cells corresponding to k sites. Each of these k sites are nearest (and equidistant) to q, and therefore, $C_P(q)$ is the largest empty circle having these k sites on its boundary.

(ii) $\rightarrow$ If a point q exists on $b(p_i, p_j)$ s.t. the boundary of $C_P(q)$ contains only p_i and p_j, then p_i and p_j are the nearest sites of q, and therefore, q lies in the Voronoi cells of (both and only) p_i and p_j. Thus, $b(p_i, p_j)$ defines an edge shared by $vc(p_i)$ and $vc(p_j)$.

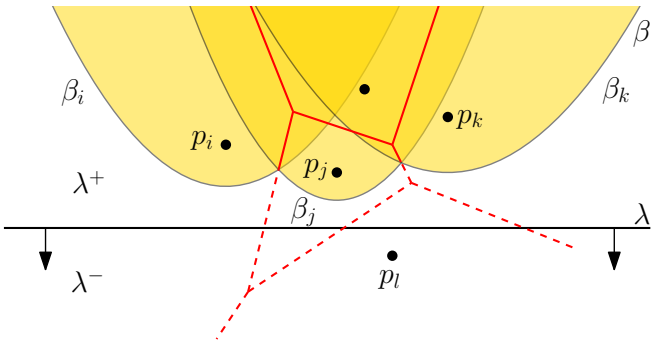
$\leftarrow$ If $b(p_i, p_j)$ defines an edge of $VD(P)$, then for any point q lying on this edge, p_i and p_j are the nearest sites. Hence, the boundary of $C_P(q)$ contains only p_i and p_j. □

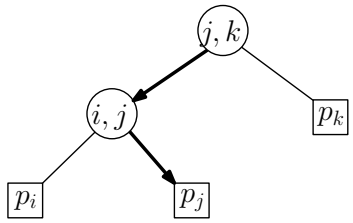
4.2.2 Fortune's Algorithm

Fortune's algorithm[1] uses the plane sweep strategy in which a horizontal sweep line λ moves from top to bottom along the xy plane and maintains the information about the part of the $VD(P)$ above λ that can never change whatever may be the distribution of the sites below λ. To know which part of $VD(P)$ above λ remains fixed irrespective of the sites below λ, we need the following definition and subsequent theorems.

Definition 4.2.3 (beach line) *Let λ^+ be the closed half-plane above λ, and λ^- be the remaining half-plane (Fig. 4.7). If a point q in λ^+ is at least as near to some site p_i in λ^+ as q is to the sweep line λ, then q lies in $vc(p_i)$—a fact already decided, and decided for ever—as sites in λ^- cannot alter this fact. Such a point q lies inside the parabolic region defined with p_i as the*

[1] Steve Fortune. A sweepline algorithm for Voronoi diagrams. Algorithmica, vol. 2, pages 153–174, 1987.





Search path (thick directed edges) in T to find the arc lying above p_l

Figure 4.7: The beach line β formed by the sequence of three parabolic arcs, $\beta_i, \beta_j, \beta_k$, corresponding to the three sites, p_i, p_j, p_k, respectively. β_i is the parabolic arc with p_i as the focus of the parabola and the sweep line λ as the directrix. Similarly, for β_j and β_k, the directrix is λ and the respective foci are p_j and p_k. The point of intersection between β_i and β_j traces one edge of $VD(P)$ and that between β_j and β_k traces another. The portions of these two edges (shown in red and bold) are confirmed for ever and will not be changed by sites lying below λ whatsoever. The dashed portions above λ may change depending on positions of sites lying below λ; e.g., the edge traced by the intersection point between β_j and β_k is intersected by two edges at a point (vertex of $VD(P)$) that lies above λ.

*focus and λ as the directrix. Hence, the union of all the parabolic regions with the sites in λ^+ as foci and λ as the directrix contains the part of $VD(P)$ that cannot get changed by the sites in λ^-. This part of $VD(P)$ is bounded from below by a sequence of parabolic arcs, which is called the **beach line**, denoted by β.*

Theorem 4.2.4 (beach line) *β is x-monotone, i.e., if we move along β from left to right, then the x-coordinate always increases (although the y-coordinate may increase or may decrease).*

Theorem 4.2.5 (event points) *The structure of β changes as the sweep line moves down, and in this process, new parabolic arcs appear (**site event**) and existing parabolic arcs disappear (**circle event**).*

Theorem 4.2.6 (breakpoint) *A **breakpoint**, given by the point of intersection between two consecutive parabolic arcs of β, traces an edge of $VD(P)$.*

4.2.3 Information about the Beach line

The information about β is stored in an AVL tree, namely T, in terms of the breakpoints of β. Hence, T is modified only if a breakpoint disappears from or appears in β. Such an appearance or a disappearance of a breakpoint in or from the beach line is associated with a certain ***event***, defined as follows (also see Theorem 4.2.5).

Definition 4.2.4 (events) *When the sweep line λ encounters a new site, say p_i, it is called a **site event**. A site event marks the appearance of a new parabolic arc (with p_i as the focus and*

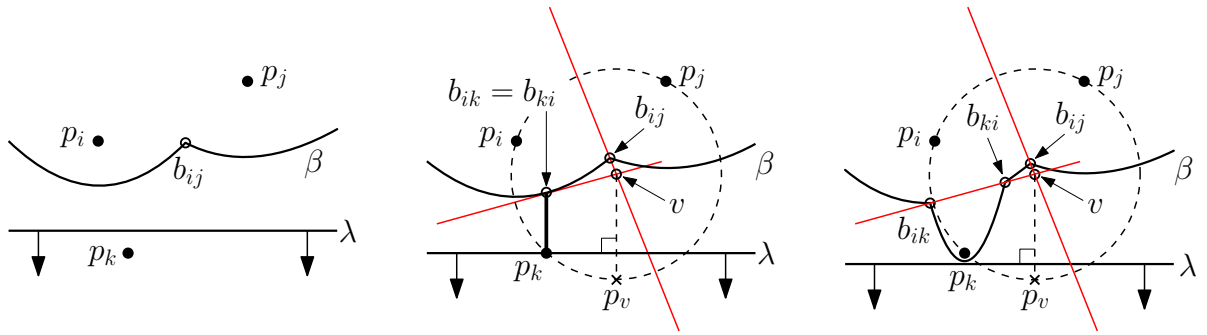


Figure 4.8: Change in β with the site event p_k. **Left:** β consists of two parabolic arcs, β_i and β_j, with p_i and p_j as the respective foci, and b_{ij} as the breakpoint. **Middle:** The site event has occurred; the degenerate parabolic arc β_k of zero width with p_k as the focus penetrates β_i lying above it. The circle event (p_v) is stored in Q (explained later, see Fig. 4.11). **Right:** Widening of β_k as λ moves down; the breakpoints b_{ik} and b_{ki} move apart.

λ as the directrix) in the beach line (Fig. 4.8).
When λ touches the lowest point of the circle passing through three sites, say p_i, p_j, and p_k, defining three consecutive arcs (β_i, β_j, and β_k) in β, it is called a ***circle event****. A circle event marks the disappearance of the second arc (i.e., β_j) among three consecutive arcs from β (Fig. 4.9).*

4.2.4 Necessary data structures

(i) AVL tree, T: A non-leaf node stores the ordered pair of sites corresponding to a breakpoint, and the leaf nodes store the sites acting as the foci of the arcs of β ordered from left to right (possible due to x-monotonicity of β: Theorem 4.2.4).

(ii) Priority queue, Q: An event queue that initially stores the site events (coordinates of the sites, sorted w.r.t. y-coordinates), and stores the circle event (y-coordinate of λ when the circle event occurs). Q may be implemented as a heap or an AVL tree.

(iii) Doubly connected edge list (DCEL), L: Stores the vertices, edges, and faces of $VD(P)$ as the output. The half-infinite edges are made finite in the DCEL by considering a sufficiently large bounding box.

4.2.5 Time and space complexities

Number of circle events processed is bounded by $2n-5$ (Theorem 4.2.2). Detection and insertion of a circle event in Q takes $O(\log n)$ time. For a site event, searching in T with necessary update in β takes $O(\log n)$ time. Insertion of a new vertex and a new edge in DCEL takes $O(1)$ time. Total number of vertices, edges, and faces is $O(n)$ (Theorem 4.2.2). Hence, time complexity is $O(n \log n)$.

Q always stores the sites that are yet to be handled, and the circle events resulting from only the triplets of consecutive arcs in β. The beach line beta has $O(n)$ parabolic arcs in sequence. So T stores $O(n)$ sites as foci of parabolic arcs in β, and the breakpoints for each doublet of consecutive arcs in β. Hence, space complexity is $O(n)$.

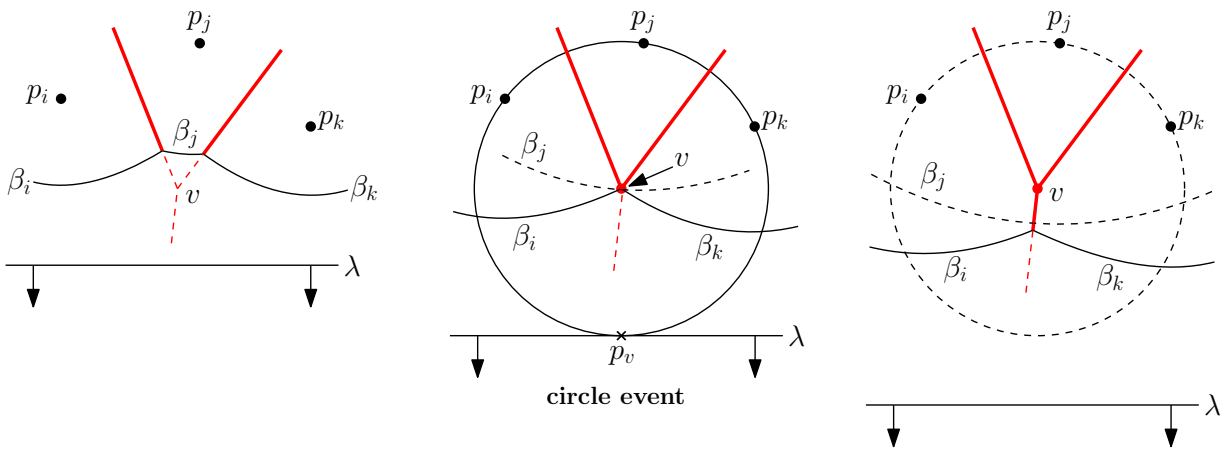
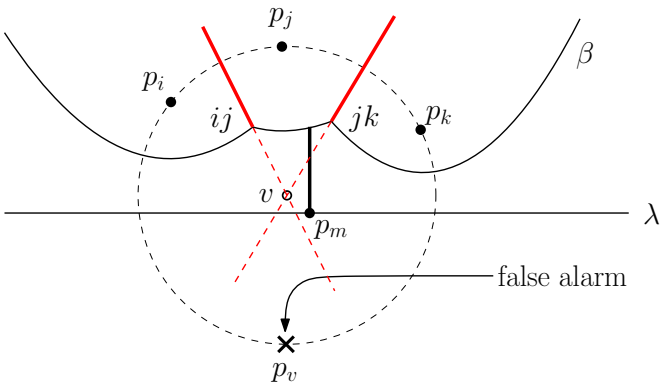


Figure 4.9: Disappearance of a parabolic arc from β with the circle event p_v. **Left:** β consists of three parabolic arcs, β_i, β_j and β_k, with p_i, p_j, and p_k as the respective foci. The two breakpoints trace two edges of $VD(P)$, shown in red. **Middle:** The circle event has occurred as λ encounters p_v (the lowermost point of $C_P(v)$). The point v becomes a vertex of $VD(P)$. **Right:** The parabolic arc β_j has disappeared from β as λ moves down from v, and the third edge emerging from v is now being traced by the breakpoint between β_i and β_k.

Figure 4.10: An instance of ***false alarm*** p_v corresponding to an intersection point v between the perpendicular bisectors traced by the breakpoints ij and jk. This is a circle event because λ encounters another site p_m before p_v, which prevents v to be a vertex of $VD(P)$. The circle event p_v stored as a node ν_v in Q is deleted from Q in constant time (with another $O(\log n)$ time for re-balancing or re-heapification) using the pointer from β_j to ν_v. Had the site p_m been below p_v, v could have been a vertex of $VD(P)$.



Algorithm 13: VORONOIDIAGRAM(P)

```
1 T ← ∅, Q ← {site events}, DCEL D ← ∅
2 while Q ≠ ∅ do
3     q ← Dequeue(Q)
4     if q is a site event then
5         HandleSiteEvent(q)
6     else
7         HandleCircleEvent(q)
8 Compute the bounding box (BB) that contains all the vertices of VD(P), and attach the
  half-infinite edges to BB by properly updating the DCEL.
9 Traverse the half-edges of the DCEL to add the cell (face) records and the pointers to
  and from them.
```

Procedure HandleSiteEvent(p_m)

1 **Process T and remove false alarms.**
   i) If T is empty, then make p_m the root of T and return.
   ii) Search in T for the pair of consecutive breakpoints that gives the arc (say, β_j, as shown in Fig. 4.11) of β vertically above p_m.
   iii) If the leaf node ν_j (corresponding to p_j) in T points to some circle event in Q, then the circle event is deleted from Q, as it is a ***false alarm*** (Fig. 4.10).
   iv) Replace ν_j by a subtree T_m having the following nodes (Fig. 4.11):
      ⊙ root node ν_{jm}: represents the new left breakpoint, $\langle j, m\rangle$;
      ⊙ right child node ν_{mj} of ν_{jm}: represents the new right breakpoint, $\langle m, j\rangle$;
      ⊡ leaf node ν_{jL} as the left child of ν_{jm}: represents the left part of β_j (say β_{jL});
      ⊡ leaf node ν_m as the left child of ν_{mj}: represents the new arc β_m;
      ⊡ leaf node μ_{jR} as the right child of ν_{mj}: represents the right part of β_j (say β_{jR}).
   v) If T becomes unbalanced after the above operation, then rebalance T.

2 **Add circle events in Q.** If the two edges traced by the two breakpoints corresponding to the triplet of consecutive arcs, $\langle \beta_i, \beta_{jL}, \beta_m\rangle$, converge with the corresponding circle event point lying below λ, then store this circle event in Q, add a pointer from ν_{jL} to this event in Q and a pointer from this event to ν_{jL}. Repeat this for the triplet $\langle \beta_m, \beta_{jR}, \beta_k\rangle$, where β_k is the arc to the right of β_{jR}.

3 **Update DCEL.** Create two new half-edge records in DCEL for the edge traced by the two new breakpoints, which separates $vc(p_j)$ and $vc(p_m)$.

Procedure HandleCircleEvent(p_i)

1 **Process T.** See Fig. 4.9. Use the pointer from p_j (which is just dequeued) to find its leaf node ν_j in T and its disappearing arc β_j in β. Delete ν_j from T. Update the breakpoint info at the internal nodes of T. Rebalance T if needed.

2 **Remove false alarms from Q.** For the arc triplet in β in which β_j was the middle arc, the corresponding circle event p_j has already been deleted from Q in Step 3 of VoronoiDiagram(P). Hence, now consider the other two arc triplets—one in which β_j is the first arc, and the other in which β_j is the third arc. For these two triplets, the respective middle arcs (β_i and β_k in Fig. 4.9) correspond to the predecessor and the successor of ν_j in T. Using their pointers to circle events (if not Null) in Q, delete the respective circle events from Q.

3 **Insert new circle events in Q.** Consider the new triplets of consecutive arcs arisen out of deletion of β_j from β. Check whether their breakpoints converge, with the points of corresponding circle events lying below λ; if so, add corresponding circle events in Q and set pointers between them and their respective nodes in T (as explained in Fig. 4.11).

4 **Update DCEL.** Add the center (v in Fig. 4.9) of the circle event's circle as a vertex in DCEL. Create the half-edge records in DCEL with v as the start or the end vertex.

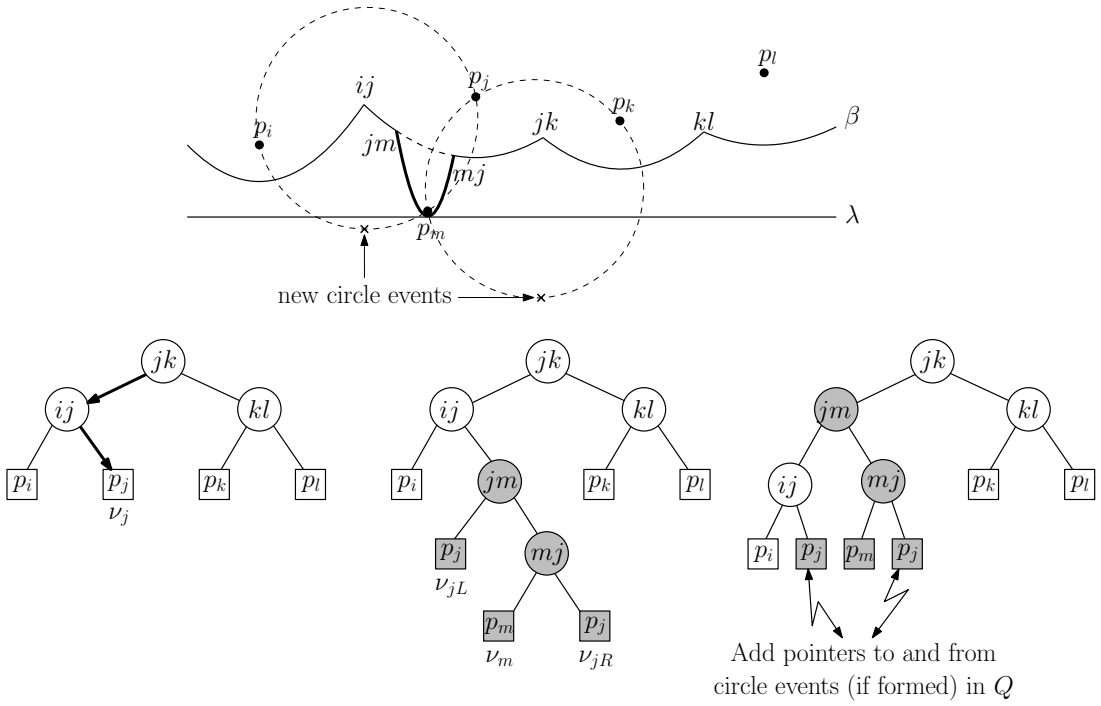


Figure 4.11: Illustration of the procedure HandleSiteEvent(p_m). **Top:** A new parabolic arc β_m (shown in bold) appears in the beach line β for the site event p_m. **Bottom-left:** Searching in T (bold and directed edges) for the node ν_j corresponding to p_j that has formed β_j lying above p_m. **Bottom-mid:** The resultant new arcs in β are $\langle p_j, p_m, p_j \rangle$, and new breakpoints are $\langle jm, mj \rangle$ (simplified representation of $\langle (p_j, p_m), (p_m, p_j) \rangle$). As T contains the sites as leaf nodes and the breakpoints as intermediate/root nodes, with the in-order property of all points stored in T, a new subtree of these five nodes (shown in gray) is inserted in $O(\log n)$ time. **Bottom-right:** Structure of T after re-balancing in $O(\log n)$ time. Three new triplets of arcs are formed: $\langle \beta_i, \beta_j, \beta_m \rangle$, $\langle \beta_j, \beta_m, \beta_j \rangle$, and $\langle \beta_m, \beta_j, \beta_k \rangle$. Each triplet corresponds to two consecutive breakpoints in β. The breakpoints jm and mj of the 2nd triplet correspond to one edge, whereas those of the 1st and the 3rd triplets correspond to two edges each. If the two edges for the 1st triplet are converging and give rise to a circle event (Fig. 4.9) lying below λ, then this new circle event is inserted in Q. And in the node ν_{jL} of p_j in the first triplet, the pointer to this circle event is added. A pointer from this circle event to ν_{jL} is also added. Similar operations are performed for the node ν_{jR} of p_j in the 3rd triplet.

4.2.6 Handling degeneracies

If four sites are co-circular, then two identical vertices are inserted in DCEL, with a zero-length edge between them (Fig. 4.12:left). A post-processing can be done to detect and rectify this in DCEL.

If a site p_k lies exactly below a breakpoint ij, then it splits each of β_i and β_j into two arcs while inserting β_k in β. The right part of β_i and the left part of β_j after such splitting are of zero-length (Fig. 4.12:right). The newly formed arc triplet in which the zero-length part of β_i is the middle arc defines a circle event. This circle event is centered at the current breakpoint ij, with its bottommost point coinciding with p_k, which is the event point just dequeued from Q. Just after p_k is processed as a site event, the circle event p_k is processed, and the vertex at

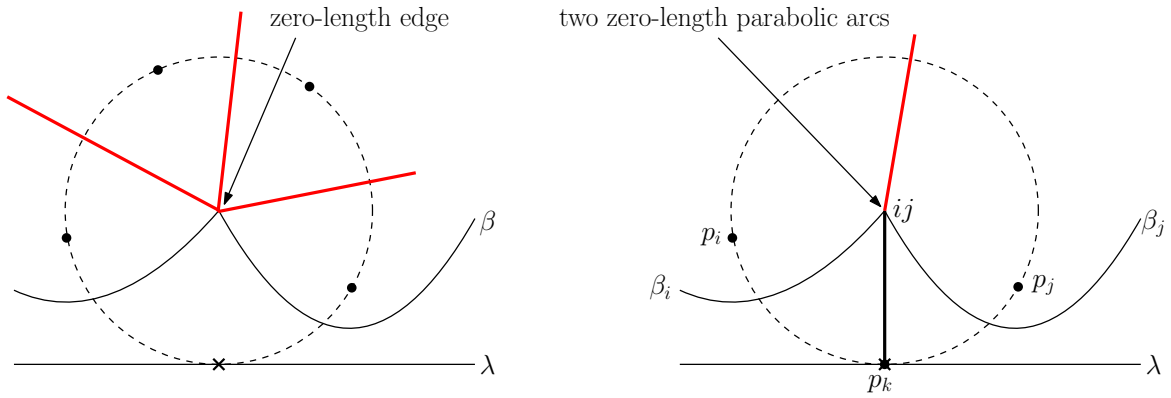


Figure 4.12: Degeneracies (Sec. 4.2.6). **Left:** Four sites are co-circular. **Right:** The new site lies vertically exactly below a breakpoint.

ij is correctly inserted in DCEL. The zero-length part of β_i is deleted from β, i.e., T. Similar handling for the triplet in which the zero-length part of β_j is the middle arc, is done by the algorithm, forming a duplicate vertex of $VD(P)$ at ij. The duplicate vertices with the edge of zero-length between them in DCEL can be post-processed to obtain the minimality of DCEL.

4.2.7 DCEL

Doubly connected edge list (DCEL) is a data structure to store and represent any planar subdivision, such as Voronoi diagram, planar graph, etc. It can also be used to represent the triangulated surface of any polyhedron. It captures the topological relation among vertices, edges, and faces, and hence finds application in many geometric and graph algorithms.

A DCEL contains three lists: one for vertices, one for edges, and one for faces. Each record of a list may contain some auxiliary information. For example, the record corresponding to each face in the face list may contain the name of the area or a pointer to many attributes related to that area. As each edge usually bounds two faces, it is considered as two *half-edges* in the edge list, one being the *twin* of the other. Owing to this, the end vertex of a half-edge is the same as the start vertex of its twin. Each half-edge e is incident on a single face f, and so the record of e in the edge list contains the ID of f. The record of e also contains the ID of the next half-edge and that of the previous half-edge of f. To reach any face f' adjacent to f, we can use the edge e incident on f to know the twin of e, and then start from twin(e) and subsequently traverse all the edges of f'.

Following are the information commonly stored in the DCEL for vertices, edges, and faces (Fig. 4.13).

- Vertex: ID, (x, y) coordinates, ID of any outgoing edge.
  For example, in Fig. 4.13, out of the three edges outgoing from u, we can store just the ID of $e(u, v)$.
- Half-edge: ID, start vertex ID, twin edge ID, incident face ID, next edge ID, previous edge ID.
  For example, in Fig. 4.13, for the half-edge $e(u, v)$, we have to store the ID of u, ID of $e(v, u)$, `1` as the associated face ID, etc.

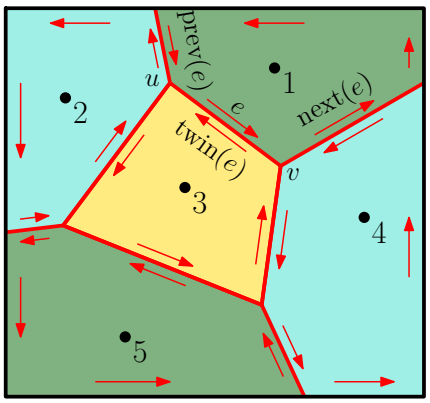


Figure 4.13: DCEL storing the Voronoi diagram of five sites with its bounding box.

- Face: ID and the ID of any incident edge.
  For example, in Fig. 4.13, for face of site `1`, we can store just the ID of $e(u, v)$; similarly, for face of site `3`, we store the ID of $e(v, u)$.

To prepare the DCEL corresponding to a Voronoi diagram, vertex and edge records are first prepared while Algorithm 13 runs. Then for each face, one of its incident edges is stored from the edge information. This is done by considering all the edges in the edge list one by one.

Chapter 5

Randomized Algorithms

Creativity is the ability to introduce order into the randomness of nature.

— **Eric Hoffer**
(Social Philosopher, US)

The image aside is an artwork created by a circle packing algorithm recently designed by us. It is implemented by Sourav De, who graduated from our department in 2013. The concerned research paper titled ***Digital Circlism as Algorithmic Art*** has been presented by us in 9th International Symposium on Visual Computing (ISVC 2013), held in Crete, Greece in July, 2013.
To see more, visit:
http://cse.iitkgp.ac.in/~pb/research/circlism

A ***randomized algorithm*** adds a degree of randomness during its execution, expecting to achieve a good performance in the average case over all possible random choices. When a randomized algorithm uses a random input to reduce the expected runtime or memory usage, but always terminates with a correct result within a bounded amount of time, it is called a ***Las Vegas algorithm***. On the contrary, an algorithm, which terminates in polynomial time, but depending on the random input, stands a chance of producing an incorrect result, is said to be a ***Monte Carlo algorithm***. In other words, a Las Vegas algorithm is randomized in the sense that its runtime varies randomly, as it terminates only on producing the correct answer. But a Monte Carlo algorithm, although has a deterministic runtime, can produce output that may be incorrect with a certain (typically small) probability.

For implementation, a randomized algorithm is approximated using a pseudo-random number generator in place of a true source of random bits; such an implementation may somewhat deviate from the expected theoretical behavior. The pseudo-random numbers are used by the randomized algorithm in addition to the input. During execution, it takes random choices depending on these random numbers. The behavior (output) can vary if the algorithm is run multiple times on the same input.

The process of removing randomness as much as possible is called ***derandomization***. Methods employed for derandomization are usually based on conditional probabilities, discrepancy theory, expander graphs, etc.

5.1 Basics on Probability and Expectation

In an experiment, each distinct outcome is called a ***sample point***. The set comprising all sample points together is called ***sample space***, usually denoted by Ω. For example, consider an experiment of n coin tosses. Here, the sample space is $\{H,T\}^n$, which has 2^n distinct sample points. For $n = 1$, $\Omega = \{H,T\}$; for $n = 2$, $\Omega = \{HH,HT,TH,TT\}$; for $n = 3$, $\Omega = \{HHH,HHT,HTH,HTT,THH,THT,TTH,TTT\}$; and so on.

An ***event*** E is a subset of Ω. For example, in the above coin-toss experiment, the set of coin tosses with even number of `heads` is an event. ***Probability function*** $\mathbf{P}$ is a function from Ω to $[0,1]$ such that $\sum_{x\in\Omega} \mathbf{P}(x) = 1$. Probability of an event E is $\sum_{x\in E} \mathbf{P}(x)$. For two independent events A and B, $\mathbf{P}(A \cap B) = \mathbf{P}(A) \times \mathbf{P}(B)$.

5.1.1 Random Variable

A ***random variable*** $X : \Omega \to \mathbb{R}$ is a function whose value is subject to variations due to chance or randomness, in a mathematical sense. As opposed to other mathematical variables, a random variable cannot have a single, fixed value, but can take on a set of possible different values, each with an associated probability. The random variable X can assume the value x with probability $\mathbf{P}(X = x)$, and hence the ***expectation*** (i.e., the *expected* or the *weighted average* value) of X is given by

$$\mathbf{E}(X) = \sum_{x\in\Omega} x \cdot \mathbf{P}(X = x). \tag{5.1}$$

Example: The expected number of `heads` when a fair coin is tossed n times, is $n/2$. To show this, define X to be a random variable that gives the number of `heads`. Observe that $X = i$ means we have i `heads` out of n trials, which is possible in $\binom{n}{i}$ different ways. So, $\mathbf{P}(X = i) = \binom{n}{i}/2^n$, and hence from Eqn. 5.1,

$$\begin{aligned}
E(X) &= \sum_{i=0}^{n} i \cdot \mathbf{P}(X = i) = \sum_{i=0}^{n} i \cdot \frac{\binom{n}{i}}{2^n} = \frac{1}{2^n} \sum_{i=0}^{n} i\binom{n}{i} \\
&= \frac{1}{2}\frac{1}{2^n}\left(\sum_{i=0}^{n} i\binom{n}{i} + \sum_{i=0}^{n}(n-i)\binom{n}{n-i}\right) \quad \triangleright \text{writing again in reverse order} \\
&= \frac{n}{2^{n+1}} \sum_{i=0}^{n} \binom{n}{i} \quad \triangleright \text{since } \binom{n}{i} = \binom{n}{n-i} \\
&= \frac{n}{2^{n+1}} 2^n \\
&= n/2.
\end{aligned}$$

5.1.2 Linearity of Expectation

Theorem 5.1.1 *If X and Y are two (dependent or independent) random variables, and a and b are two real numbers, then $\mathbf{E}(aX + bY) = a\mathbf{E}(X) + b\mathbf{E}(Y)$.*

The proof is as follows.

$$\begin{aligned}\mathbf{E}(aX+bY) &= \sum_{x\in\Omega}\sum_{y\in\Omega}(ax+by)\mathbf{P}(X=x,Y=y)\\ &= a\sum_{x\in\Omega}x\sum_{y\in\Omega}\mathbf{P}(X=x,Y=y)+b\sum_{y\in\Omega}y\sum_{x\in\Omega}\mathbf{P}(X=x,Y=y)\\ &= a\sum_{x\in\Omega}x\mathbf{P}(X=x)+b\sum_{y\in\Omega}y\mathbf{P}(Y=y)\\ &= aE(X)+bE(Y).\square\end{aligned}$$

A general result based on Theorem 5.1.1 is that, for n random variables, namely $X_1, X_2, \ldots, X_n$, and n real numbers, namely $a_1, a_2, \ldots, a_n$, we get $\mathbf{E}\left(\sum_{i=1}^{n} a_i X_i\right) = \sum_{i=1}^{n} a_i E(X_i)$.

Theorem 5.1.1 (Linearity of Expectation) can be used to solve the previous problem of finding the expected number of `heads` when a fair coin is tossed n times. As before, let X be the number of `heads` out of n tosses. Now, for $1 \leqslant i \leqslant n$, define X_i as a binary random variable, which gets 1 if the ith toss results in `heads` and 0 if it is `tails`. Then $\mathbf{E}(X_i) = 1 \cdot \mathbf{P}(X_i = 1) + 0 \cdot \mathbf{P}(X_i = 0) = 1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{2} = \frac{1}{2}$. As $X = \sum_{i=1}^{n} X_i$, we get $\mathbf{E}(X) = \sum_{i=1}^{n} \mathbf{E}(X_i) = n \times \frac{1}{2} = n/2$.

5.2 Las Vegas Quick Sort

We know how to construct a height-balanced BST, e.g., AVL tree or median-pivoted BST (constructed recursively using the median as the pivot at every stage). It is true that all these algorithms take $O(n \log n)$ runtime which is optimal, but the algorithms are inherently complex and require expert coders to implement, with many lines of code. As an alternative, we can design a much simpler algorithm for constructing a BST whose *expected height* would be $O(\log n)$, with an *expected runtime* for construction as $O(n \log n)$. In fact, the same idea can be used to enhance the deterministic Quick Sort algorithm (worst-case runtime is $O(n^2)$) to a randomized Quick Sort with *expected runtime* as $O(n \log n)$. This section sketches this based on the idea of ***central splitter***.

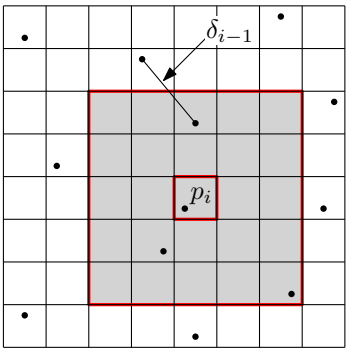
Let A be the given unordered list of n elements. Let A' denote the sorted list containing the elements of A. We define $p \in A$ to be a central splitter of A if p occurs anywhere in the middle 50% elements of A'. In other words, p is a central splitter if at least 25% elements of A are smaller than p and at least 25% elements of A are greater than p. Now, it is easy to see that if a central splitter is used as the pivot (recursively) while partitioning A during the Quick Sort, then the partition ratio will range from 1:3 to 3:1, thus yielding the runtime recurrence as $T(n) = T(n/4) + T(3n/4) + O(n)$, which implies $T(n) = O(n \log n)$. The procedure to find a central splitter is explained below.

Pick an element p randomly from A. Check whether p is a central splitter by comparing all elements of A with p. This can be done in $O(n)$ time. If p is a central splitter, then use p as the pivot of Quick Sort (or root of BST); otherwise attempt once more. As each element of A is equally likely to be picked as p, and as a central splitter can be from 50% elements of A, the probability that p is a central splitter is $\frac{1}{2}$. Hence, the expected number of trials to find a central splitter is 2. Thus the central splitter is obtained in $O(n)$ time, which is subsumed in the partition time!

5.3 A Las Vegas Algorithm for Closest Pair

Let $P = \{p_1, p_2, \ldots, p_n\}$ be the given point set on 2D plane. Our task is to find a/the closest pair of points in P. Iteratively, we find the closest pair in $P_i = \{p_1, p_2, \ldots, p_i\}$ using the closest pair already obtained in P_{i-1} and a "small neighborhood" N_i around p_i. The procedure is as follows.

Let δ be the closest-pair distance in P_{i-1}. Apply a transformation (linear scaling plus 2D translation) on P so that all points have x- and y-coordinates in the real interval $[0, 1]$. That is, all points are contained in the axis-parallel unit square A whose bottom-left corner is at $(0, 0)$. Subdivide A into sub-squares, each of size $\delta/2 \times \delta/2$. Each sub-square gets a unique ID as per its location from $(0, 0)$. Hash only those sub-squares containing points of P_{i-1} to a hash table T, based on their IDs.



Now the following pair of observations:
Obs. 1) Since the closest-pair distance in P_{i-1} is δ, no sub-square would contain more than one point from P_{i-1}. Hence, for each hashed sub-square, store also the index of the point that it contains.
Obs. 2) The closest pair in P_i would be closer than that in P_{i-1} if and only if p_i is one member of the closest pair, or, for some point $p_j \in P_{i-1}$, $d(p_i, p_j) < \delta$, and then p_j would lie in one of the 25 sub-squares whose (row-wise and column-wise) middle sub-square contains p_i.

Hence, just from the ID of the sub-square S_i containing p_i, compute the IDs of the other 24 sub-squares surrounding S_i, and then check each of the 25 sub-squares, find their existence in T, and get the corresponding points of P_{i-1}. Find the nearest to p_i among them. If the corresponding pair has distance less than δ, then only we need to redefine the sub-square decomposition of A and also the hash table T. Since the points in P are in random order, we have an expected $O(n)$ runtime, as explained below.

Let X_i be the random variable, which gets 1 if the closest pair in P_i is different from that in P_{i-1}, and 0 otherwise. From Obs. 2, $X_i = 1$ if and only if p_i is a point in the closest pair of P_i. Since each point in P_i is equally likely to be one of the two points forming the closest pair in P_i, we get

$$E[X_i] = Pr[X_i = 1] = \frac{2}{i}.$$

The major operations in each iteration of the algorithm are:
finding the nearest point of P_{i-1} from p_i out of at most 25 hashed points in T—in $O(1)$ time,
re-hashing all i points in P_i when the closest pair in P_i is different from that in P_{i-1}.
As the first closest pair is (p_1, p_2) and subsequently there are $n - 2$ iterations, the expected runtime is given by

$$T(n) = O(n) + O\left(\sum_{i=3}^{n} i \cdot E[X_i]\right) = O(n) + O(2n) = O(n).$$

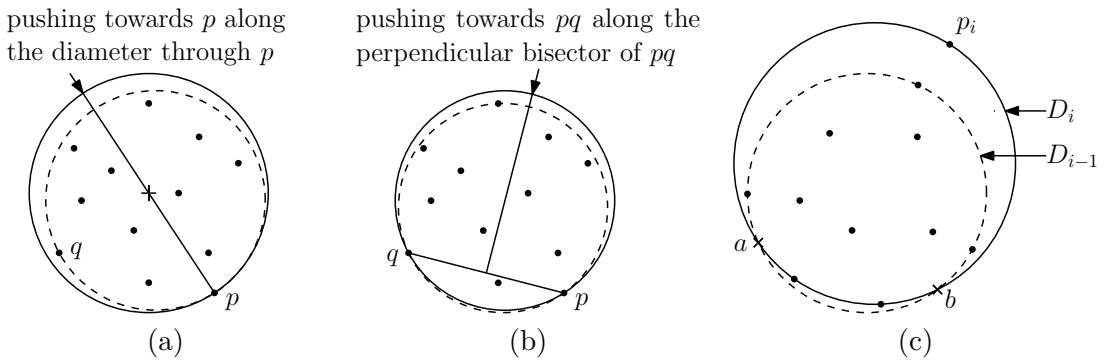


Figure 5.1: (a) We can reduce the enclosing circle passing through a single point p by shifting its center along the diameter through p, towards p, until it touches a second point. (b) When the circle passes through two points, p and q, we can reduce it further by shifting its center towards pq until it touches a third point. (c) D_{i-1} changes to a larger disc D_i when $p_i \notin D_{i-1}$. Note that the smaller arc ab of C_i lies inside D_{i-1}. Hence, the only point of P_i that can lie on the larger arc of C_i is p_i.

5.4 Minimum Enclosing Disk by Las Vegas Algorithm

Given a set of 2D points $P = \{p_1, p_2, \ldots, p_n\}$ in 2D, we have to find the disk of minimum radius that contains all points of P.

Naive algorithm: Consider each triple of points from P, and check whether every other point of P lies inside the corresponding circum-circle. Time complexity: $O(n^4)$.

Efficient algorithms: Can be found in $O(n \log n)$ time based on *farthest-point Voronoi diagram* [1].
As each vertex of such a diagram represents a circle containing all the points in P, the one with minimum radius is the answer. There is also a complicated $O(n)$-time algorithm using *linear programming* [2].

Here we discuss a ***Las Vegas algorithm*** to compute the minimum enclosing disk. First we generate a random permutation of the points in P. Let $P_i = \{p_1, p_2, \ldots, p_i\}$. Let D_i be the ***minimum enclosing disk*** (MED) of P_i, and C_i be the boundary of D_i. We call C_i as the ***minimum enclosing circle*** (MEC) of P_i.

We have the following observations, used to prove Theorem 5.4.1, which leads to an ***incremental algorithm***. Figure 5.1 shows the idea behind the above observations. The formal proof in geometry is left to you.

Observation 5.1 *For a set of points P in general position, the MED has at least three points on its boundary, or it has two points as the endpoints of its diameter. If there are three points, then they subdivide the MEC into arcs of angle at most π.*

Observation 5.2 *Given a disk of radius r_1 and a circle of radius r_2, with $r_1 < r_2$, the arc (of the circle) lying inside the disk is smaller than the arc lying outside the disc.*

Theorem 5.4.1 *If $p_i \in D_{i-1}$ then $D_i = D_{i-1}$; otherwise, $p_i \in C_i$.*

Algorithm 14: Algorithm MED(P)

Input: A set P of n points in the plane
Output: The minimum enclosing disk MED for P

```
1 Randomly permute P to get {p_1, ..., p_n}
2 D_2 ← MED(p_1, p_2) ▷ unique disk with p_1p_2 as diameter
3 for i ← 3 to n do
4     if p_i ∈ D_{i−1} then
5         D_i ← D_{i−1}
6     else
7         D_i ← MEDWith1Point({p_1, p_2, ..., p_i}, p_i)
8 return D_n
```

Procedure MEDWith1Point(P, q)

Input: A set of points P and $q \in P$
Output: MED for P with q on the boundary

```
1 Randomly permute P to get {p_1, ..., p_n}
2 D_1 ← MED(p_1, q) ▷ unique disk with p_1q as diameter
3 for i ← 2 to n do
4     if p_i ∈ D_{i−1} then
5         D_i ← D_{i−1}
6     else
7         D_i ← MEDWith2Points({p_1, p_2, ..., p_i}, p_i, q)
8 return D_n.
```

Procedure MEDWith2Points(P, p, q)

Input: A set of points P and $p, q \in P$
Output: MED for P with p and q on the boundary

```
1 Randomly permute P to get {p_1, ..., p_n}
2 D_0 ← MED(p, q) ▷ unique disk with pq as diameter
3 for i ← 1 to n do
4     if p_i ∈ D_{i−1} then
5         D_i ← D_{i−1}
6     else
7         D_i ← Disk(p, q, p_i) ▷ unique disk with p, q, p_i on its boundary
8 return D_n.
```

Proof: Based on Observations 5.1 and 5.2; see also Fig. 5.1. Suppose $p_i \notin D_{i-1}$ and also $p_i \notin C_i$. Let r_1 = radius of D_{i-1} and r_2 = radius of D_i. Using Observation 5.2, C_i intersects D_{i-1} in an arc (of C_i) with angle (subtended at the center of C_i) less than π. Since $p_i \notin C_i$, points defining D_i should lie in the said arc (the smaller arc ab in Fig. 5.1c) implying an angle more than π for the larger arc, which contradicts Observation 5.1.

The algorithm: If the new point $p_i \notin D_{i-1}$, then we need to find D_i where p_i is constrained to lie on C_i. Recursively, if we encounter a point p_j that lies outside the current disk, then we recurse on a subproblem where the two points p_i and p_j are constrained to lie on the boundary. How long can this go on? In the next level, we have three points constrained to lie on the boundary and that defines a unique disk! See Algorithm 14 and its demonstration in Figure 5.2.

5.4.1 Time Complexity

With the nested recursion, the worst-case time complexity is $O(n^3)$.
For the expected case:
MEDWITH2POINTS needs $O(n)$ time.
MEDWITH1POINT needs $O(n)$ time, since MEDWITH2POINTS is called at most once with a low probability for every new point, as explained below by ***backward analysis***.

Take a subset $P_i = \{p_1, p_2, \ldots, p_i\}$ whose MED is D_i by definition. If p_i is removed from P_i, and if $p_i \in D_i - C_i$, then D_i does not change. However, if $p_i \in C_i$, then D_i may change if C_i contains exactly three points one of which is p_i. (If there are more than three points on C_i, then D_i does not change on removing p_i.) One of the boundary points is known to be $q \in P_i$. So, the probability that p_i is one of the other two boundary points is $2/(i-1)$. Hence, the probability of calling MEDWITH2POINTS from MEDWITH1POINT (when p_i is the current point) is $\frac{2}{i-1}$; and thus the expected runtime of MEDWITH1POINT becomes

$$O(n) + \sum_{i=2}^{n} O(i) \times \frac{2}{i-1} = O(n).$$

Similarly, the expected runtime of MED can also be shown to be $O(n)$.

References

1. M.I. Shamos and D. Hoey, Closest Point Problems, in Proc. *16th IEEE Symp. Foundations of Computer Science*, pp. 151–162 (1975).
2. N. Megiddo, Linear-Time Algorithms for Linear Programming in $\mathbb{R}^3$ and Related Problems, *SIAM J. Computing: 12*, pp. 759–776 (1983).

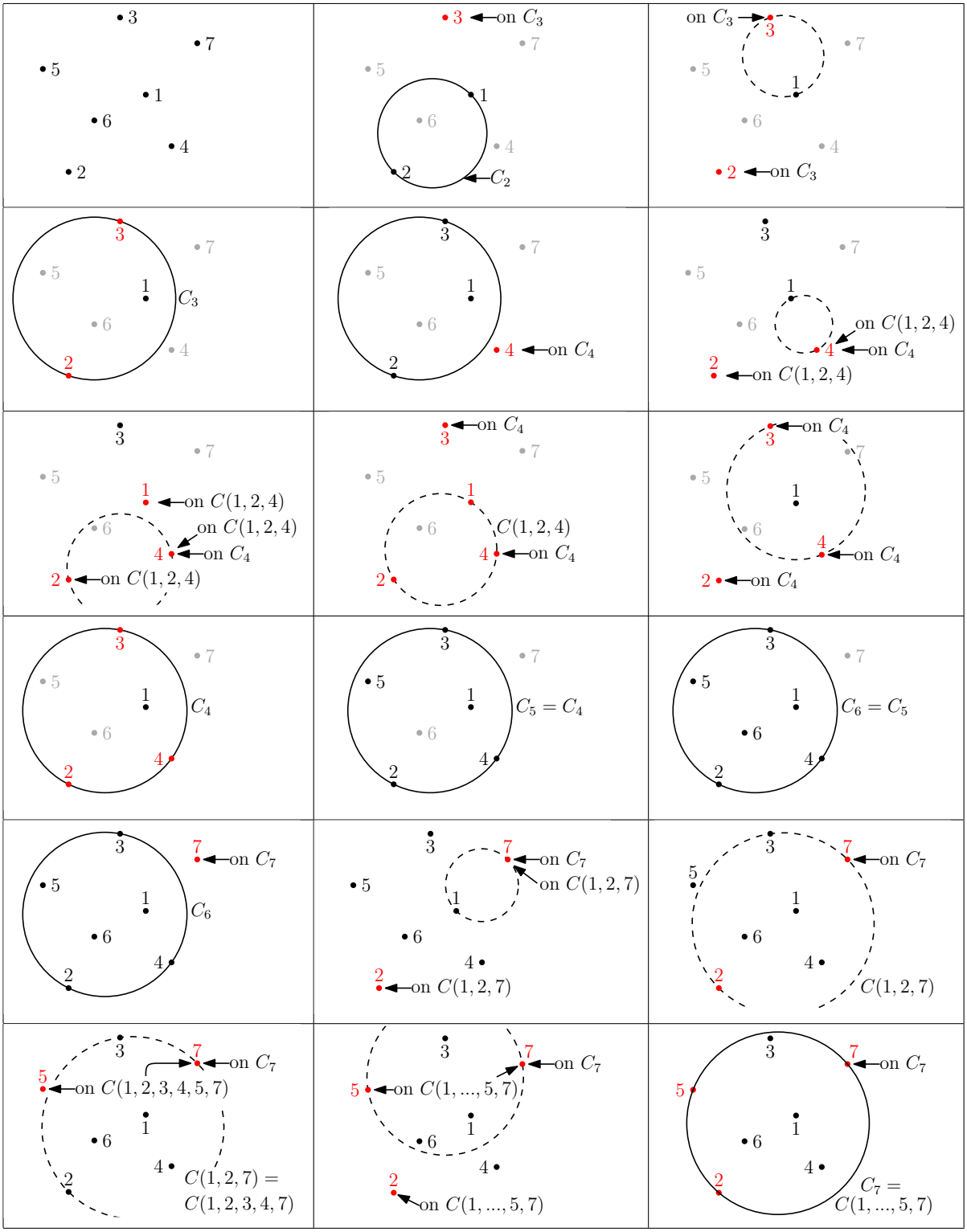


Figure 5.2: A demo of Algorithm 14 on a point set with 7 points. Note that the point set here is randomly permuted only once in the very first step. But as per the actual algorithm, every point subset is randomly permuted in every call of MEDWith1Point and MEDWith2Points. We have not done it here to just for sake of simplicity in indexing. In this demo, C_4 means MEC of $\{p_1, p_2, p_3, p_4\}$, but $C(1, 2, 4)$ means MEC of $\{p_1, p_2, p_4\}$, etc.

5.5 A Monte Carlo Algorithm for Min Cut

A ***cut*** (S, T) in an undirected graph $G = (V, E)$ is a partition of V into two non-empty sets, S and T, such that $S \cup T = V$ and $S \cap T = \emptyset$; the ***size*** of the cut is the number of *crossing edges*, given by

$$size(S, T) = |\{(u, v) \in E : u \in S, v \in T\}|.$$

A ***global min-cut*** (or minimum cut) is a cut for which the cut size is minimum over all possible cuts of G. A min s-t ***cut*** is a global min-cut in which for a given pair of vertices (s, t), there is an additional requirement that $s \in S$ and $t \in T$. Every global cut is an s-t cut for some $(s, t) \in V^2$. Thus, the min-cut problem can be solved in polynomial time by iterating over all choices of $(s, t) \in V^2$, $s \neq t$, and solving the resulting min s-t cut problem using the *max-flow min-cut theorem* and the polynomial-time *Ford-Fulkerson algorithm*, though it is not optimal. A better algorithm with $O(mn + n^2 \log n)$ time complexity has been proposed by Stoer and Wagner (1997)[1].

5.5.1 Randomized Contraction

The algorithm is proposed by D. Karger[2]. It uses the technique of ***edge contraction*** in which an edge $e(u, v) \in E$ is randomly selected and contracted to the effect that the vertices u and v are merged and replaced by a single node $\{u, v\}$; each edge originally incident at u or v is now made to incident at the new node, i.e., $\{u, v\}$. Thus, a node basically represents a subset of V, and there can be more than one edge between two such nodes. (In fact, such a graph is called *multi-graph*.) The process is repeated until there are exactly two vertices in the resultant graph; the set of edges between these two vertices is the *cut set*. See the demo in Fig. 5.3. Its min-cut size is 2, which is actually obtained; some other randomized run may not produce the optimal solution.

Success probability: Let $|V| = n$, and let C denote the set of edges of a specific min-cut of size k. Observe that no vertex in G has degree less than k, since otherwise a minimum-degree vertex would induce a smaller cut. Thus, $|E| \geqslant nk/2$. Let X_i denote the event that no edge of C is chosen during the ith ($i = 1, \ldots, n-2$) step of edge contraction. The probability that one of k edges of C is chosen in the first iteration is $k/|E| \leqslant 2/n$. Hence,

$$\mathbf{P}[X_1] \geqslant 1 - \frac{2}{n}.$$

In the second step, there are at least $(n-1)k/2$ edges, as there are $n-1$ vertices, each with degree at least k. So, the probability that one of k edges of C is chosen in the second iteration is $k/((n-1)k/2) \leqslant 2/(n-1)$. Hence,

$$\mathbf{P}[X_2|X_1] = 1 - \frac{k}{|E|} \geqslant \frac{2}{n-1}.$$

In general, at the ith iteration, there are at least $(n-i+1)k/2$ edges; so,

$$\mathbf{P}\left[X_i| \cap_{j=1}^{i-1} X_j\right] \geqslant 1 - \frac{2}{n-i+1}.$$

[1] M. Stoer and F. Wagner. A simple min-cut algorithm. *Journal of the ACM: 44*(4):585–591, 1997.

[2] D. Karger. Global Min-cuts in $\mathcal{RNC}$ and Other Ramifications of a Simple Min-cut Algorithm. *Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 21–30, 1993.

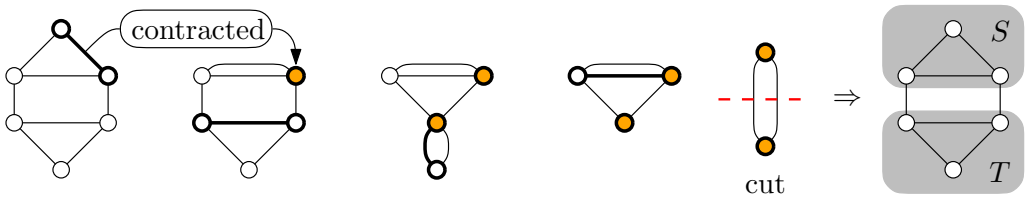


Figure 5.3: Demo of randomized algorithm to find the min-cut, which is optimal for this run. A saffron-colored vertex denotes the one obtained by edge contraction; the contracted edge is shown as a thick line.

Hence, the probability of finding a min-cut is

$$\begin{aligned}\mathbf{P}\left[\cap_{i=1}^{n-2} X_i\right] &= \mathbf{P}\left[X_1\right] \cdot \mathbf{P}\left[X_2 | X_1\right] \cdot \mathbf{P}\left[X_3 | X_1 \cap X_2\right] \cdots \mathbf{P}\left[X_{n-2} | \cap_{i=1}^{n-3} X_i\right] \\ &\geqslant \prod_{i=1}^{n-2} \left(1 - \tfrac{2}{n-i+1}\right) \\ &= \prod_{i=1}^{n-2} \tfrac{n-i-1}{n-i+1} = \left(\tfrac{n-2}{n}\right)\left(\tfrac{n-3}{n-1}\right)\left(\tfrac{n-4}{n-2}\right)\left(\tfrac{n-5}{n-3}\right)\cdots\left(\tfrac{3}{5}\right)\left(\tfrac{2}{4}\right)\left(\tfrac{1}{3}\right) = \tfrac{2}{n(n-1)} \\ &\geqslant \tfrac{2}{n^2}.\end{aligned}$$

5.5.2 Repeated Randomized Contraction

If we repeat the procedure of edge contraction, as explained in Sec. 5.5.1, for $n^2/2$ times, then the probability of not getting a min-cut in any of these trials is at most

$$\left(1 - \frac{2}{n^2}\right)^{n^2/2} < \frac{1}{e},$$

which implies that the probability of getting a min-cut is at least $1 - \frac{1}{e}$, which is reasonably good for a randomized algorithm.

Time complexity: Each contract needs $O(n)$ time. Hence, for $n-2$ contracts, we need $O(n^2)$ time. For n^2 runs, therefore, we need $O(n^4)$ time so that the success probability is reasonably good.

5.6 A Faster Min-Cut Algorithm

In the algorithm discussed in Sec. 5.5, the chance that C survives during all $n-2$ contracts, is really very small. This 'small chance' is caused by the later contractions, since the probability that some edge of a min-cut C is contracted at the ith step, goes on increasing with i. To take care of this, the algorithm of random contraction is modified as follows. Perform two independent contraction sequences to obtain G_1 and G_2 from G, each containing just as few vertices as possible, so that the expected number of graphs—out of the two contracted graphs, G_1 and G_2—in which C survives is at least 1. This implies that the probability that C survives in one of them should be at least $1/2$. Once C survives in G_1 or G_2, we can recurse on them

Algorithm 15: FastMinCut

Input: $G(V, E)$
Output: A cut C

```
1 if |V| ⩽ 6 then
2     compute min-cut C by brute force and return C
3 else
4     t ← ⌈1 + |V|/√2⌉
5     Apply two independent runs of contract to get G1 and G2 with t vertices each
6     C1 ← FastMinCut(G1)
7     C2 ← FastMinCut(G2)
8     return C as the smaller set between C1 and C2
```

with the expectation that C will survive down the recursion tree! So recurse on each of G_1 and G_2 until the number of vertices in an instance of the contracted graphs is reasonably small. Thus we get many instances of cuts, one of which can be a min-cut with a high probability. See Algorithm 21 for the steps.

To show that the algorithm is improved, we first make the following lemma.

Lemma 5.6.1 *After $n - t$ contracts, the probability that a specific min-cut C survives, is*

$$\mathbf{P}(C \text{ survives}) \geqslant \prod_{i=1}^{n-t}\left(1 - \frac{2}{n-i+1}\right) = \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right)\cdots\left(\frac{t-1}{t+1}\right) = \frac{t(t-1)}{n(n-1)}.$$

If G_1 has t nodes (and so is G_2) with the probability of at least $1/2$ that C has survived in G_1, then the above inequality holds true for $t = \lceil 1 + n/\sqrt{2} \rceil$. Using Lemma 5.6.1, the proof is as follows.[1]

$$\begin{aligned}
\mathbf{P}(C \text{ survives}) &\geqslant \frac{(\lceil 1+n/\sqrt{2}\rceil)(\lceil 1+n/\sqrt{2}\rceil - 1)}{n(n-1)} = \frac{(\lceil 1+n/\sqrt{2}\rceil)^2 - (\lceil 1+n/\sqrt{2}\rceil)}{n(n-1)} \\
&\geqslant \frac{\left(1+n/\sqrt{2}\right)^2 - \left(2+n/\sqrt{2}\right)}{n(n-1)} = \frac{\left(n^2/2 + n/\sqrt{2} - 1\right)}{n(n-1)} \\
&\geqslant \frac{\left(n^2/2 - n/2\right)}{n(n-1)} = \frac{1}{2}.
\end{aligned}$$

Define $P(n)$ = probability that C is returned by FastMinCut. Then the probability that C is returned from the recursion subtree rooted at G_1 (or G_2) is at least $\frac{1}{2}P\left(\lceil 1 + n/\sqrt{2} \rceil\right)$. Hence, the probability that C survives neither in G_1 nor in G_2 during their subsequent contractions in the recursion (Steps 6 and 7 of Algorithm 21) is at most $\left(1 - \frac{1}{2}P\left(\lceil 1 + n/\sqrt{2} \rceil\right)\right)^2$. So, from the definition of $P(n)$,

$$P(n) \geqslant \quad 1 - \left(1 - \tfrac{1}{2}P\left(\lceil 1 + n/\sqrt{2} \rceil\right)\right)^2 = P\left(\lceil 1 + n/\sqrt{2} \rceil\right) - \tfrac{1}{4}\left(P\left(\lceil 1 + n/\sqrt{2} \rceil\right)\right)^2.$$

[1] If we take $t = \frac{n}{\sqrt{2}}$, then $\frac{t(t-1)}{n(n-1)} - \frac{1}{2} = \frac{2t^2 - 2t - (n^2 - n)}{n(n-1)} = \frac{-\sqrt{2}n + n}{n(n-1)} < 0$, and it does not help. Even if we take $t = \lceil n/\sqrt{2} \rceil$, we cannot prove that $\mathbf{P}(C \text{ survives}) \geqslant \frac{1}{2}$. But with $t = \lceil 1 + n/\sqrt{2} \rceil$, we get $\mathbf{P}(C \text{ survives}) \geqslant \frac{1}{2}$.

The recursion tree, rooted at the node corresponding to G, will have a height $h = \Theta(\log n)$. To solve the above recurrence, we do a change of variables and consider an equality. Let $p(h)$ be the lower bound on the success probability. Then $p(0) = 1$ and

$$p(h+1) = p(h) - \frac{1}{4}(p(h))^2.$$

Again we do a change of variables with $q(h) = 4/p(h) - 1$, or, $p(h) = \frac{4}{q(h)+1}$. We get

$$q(h+1) = q(h) + 1 + \frac{1}{q(h)}$$

which solves to

$$h < q(h) < h + H_{h-1} + 4,$$

where $H_{h-1} = \Theta(\log h)$ is the $(h-1)$th *harmonic number*.
Hence, $q(h) = O(h)$, or, $p(h) = \Omega(1/h)$, which implies $\boxed{P(n) = \Omega(1/\log n)}$.

Time complexity: $T(n) = 2T\left(\lceil 1 + n/\sqrt{2} \rceil\right) + O(n^2)$, or, $\boxed{T(n) = O(n^2 \log n)}$.

5.7 Monte Carlo Algorithm for Max-Cut

As stated in Section 5.5, a ***cut*** is a partition (S, T) of the vertex set V of a graph $G(V, E)$. A partition of V means $S \cup T = V$ and $S \cap T = \emptyset$. The size of a cut is given by the number of edges, each of which has one vertex in S and the other in T.

Given an undirected graph G, the ***Max-Cut*** problem is to find a cut whose size is at least the size of any other cut. It is an $\mathcal{NP}$-complete problem, i.e., it has no polynomial-time algorithm unless $\mathcal{P} = \mathcal{NP}$. However, we can get a Monte Carlo solution through Algorithm 16.

Algorithm 16: Randomized algorithm for MAX-CUT$(G(V, E))$

```
S ← T ← ∅
for each v ∈ V do
    put v randomly in S or in T
return (S, T)
```

The algorithm is so simple! But what does it guarantee? It is in the following theorem.

Theorem 5.7.1 *Algorithm 16 outputs a cut of size at least $m/2$ with probability at least $\dfrac{1}{m+2}$.*

Proof. Let C be the set of edges corresponding to the cut (S, T) produced by the algorithm. For each $e_i \in E$, $\mathbf{P}(e_i \in C) = 1/2$, since the probability that two vertices of e_i are randomly put in two different sets is $1/2$. For each $e_i \in E$, define a random variable X_i, which gets 1 if $e_i \in C$, and 0 otherwise. Then $\mathbf{E}(X_i) = 1/2$ for $i = 1, \ldots, m$, where $|E| = m$. Define another random variable $X = X_1 + \ldots + X_m$. Then by Theorem 5.1.1, $\mathbf{E}(X) = \mathbf{E}(X_1 + \ldots + X_m) = \mathbf{E}(X_1) + \ldots + \mathbf{E}(X_m) = m/2$.

To prove the second part, first note that the domain of X is the integer set $\{0, 1, \ldots, m\}$. So, $\mathbf{P}(X \geqslant m/2) = \mathbf{P}(X \geqslant \lceil m/2 \rceil) = p$ (say). Then $\mathbf{P}(X < \lceil m/2 \rceil) = 1 - p$. Now,

$$\begin{aligned}
\mathbf{E}(X) &= \sum_{i=0}^{m} i \cdot \mathbf{P}(X = i) \\
&= \sum_{i=0}^{\lceil m/2 \rceil - 1} i \cdot \mathbf{P}(X = i) + \sum_{\lceil m/2 \rceil}^{m} i \cdot \mathbf{P}(X = i) \\
&\leqslant (\lceil m/2 \rceil - 1) \sum_{i=0}^{\lceil m/2 \rceil - 1} \mathbf{P}(X = i) + m \sum_{\lceil m/2 \rceil}^{m} \mathbf{P}(X = i) \\
&= (\lceil m/2 \rceil - 1)\,(1 - p) + mp \\
&= (\lceil m/2 \rceil - 1) + (m - \lceil m/2 \rceil + 1)\,p \\
&= (\lceil m/2 \rceil - 1) + (\lfloor m/2 \rfloor + 1)\,p.
\end{aligned}$$

As $\mathbf{E}(X) = m/2$, we get

$$\begin{aligned}
(\lceil m/2 \rceil - 1) + (\lfloor m/2 \rfloor + 1)\,p &\geqslant m/2 \\
\text{or, } (\lfloor m/2 \rfloor + 1)p &\geqslant 1 - (\lceil m/2 \rceil - m/2) \geqslant 1/2 \\
\text{or, } p &\geqslant \frac{1/2}{\lfloor m/2 \rfloor + 1} \geqslant \frac{1/2}{m/2 + 1} = \frac{1}{m+2}.
\end{aligned}$$

This completes the proof. □

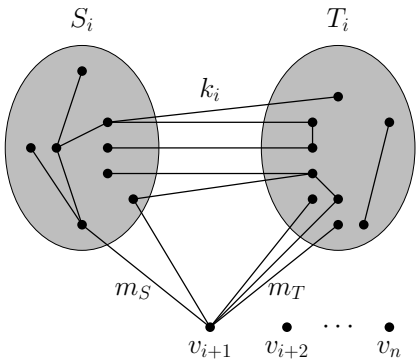


Figure 5.4: The deterministic (derandomized) way of putting a vertex v_{i+1} in S or T. Here, as $m_S < m_T$, we put v_{i+1} in S.

5.8 Derandomization of Max-Cut Algorithm

The idea of *conditional expectance* can be used for derandomization of Algorithm 16. Let $V_i = \{v_1, v_2, \ldots, v_i\} \subset V$. Let each vertex of V_i be first put in S or T in a deterministic way (explained below), and then each vertex of $V \setminus V_i$ be put in S or T by the randomized way. Let the *partial cut* corresponding to V_i be (S_i, T_i). Then the expected size of the cut (S, T), denoted by $\kappa_{(S,T)}$ and given by the expected number of crossing edges between S and T, is

$$\mathbf{E}(\kappa_{(S,T)} \mid (S_i, T_i)) = \tfrac{1}{2}\mathbf{E}(\kappa_{(S,T)} \mid (S_i, T_i) \wedge v_{i+1} \in S) + \tfrac{1}{2}\mathbf{E}(\kappa_{(S,T)} \mid (S_i, T_i) \wedge v_{i+1} \in T)$$
$$\text{or, } \max\Big(\mathbf{E}(\kappa_{(S,T)} \mid (S_i, T_i) \wedge v_{i+1} \in S), \mathbf{E}(\kappa_{(S,T)} \mid (S_i, T_i) \wedge v_{i+1} \in T)\Big) \geqslant \mathbf{E}(\kappa_{(S,T)} \mid (S_i, T_i)).$$

Hence, the deterministic way is to put v_{i+1} in S if $\mathbf{E}(\kappa_{(S,T)} \mid (S_i, T_i) \wedge v_{i+1} \in S)$ is larger than $\mathbf{E}(\kappa_{(S,T)} \mid (S_i, T_i) \wedge v_{i+1} \in T)$, or in T otherwise. Note that the partial cut (S_i, T_i) is already obtained in this deterministic way, from which we get (S_{i+1}, T_{i+1}) deterministically again! Let us denote the expected cut-size corresponding to such a deterministic policy up to v_{i+1} as $\mathbf{E}(\kappa_{(S,T)})_{i+1}$. We know that $\mathbf{E}(\kappa_{(S,T)}) \geqslant m/2$ if there is complete randomization. Hence,

$$m/2 \leqslant \mathbf{E}(\kappa_{(S,T)}) \leqslant \mathbf{E}(\kappa_{(S,T)})_1 \leqslant \mathbf{E}(\kappa_{(S,T)})_2 \leqslant \cdots \leqslant \mathbf{E}(\kappa_{(S,T)})_n.$$

This shows that the expected cut-size with a fully deterministic approach is $\mathbf{E}(\kappa_{(S,T)})_n \geqslant m/2$. And this makes the *derandomization principle*.

To find which one between S and T is the right set for v_{i+1}, we just have to compare the number of edges between v_{i+1} and vertices of S_i (put from V_i to S) with that between v_{i+1} and vertices of T_i (Fig. 5.4). To show why, let m_S and m_T be the respective numbers of these two types of edges. Let k_i be the number of crossing edges in (S_i, T_i) and m_i be the number of non-crossing edges in (S_i, T_i). In Fig. 5.4, for example, $k_i = 5$ and $m_i = 4 + 3 = 7$. Now, observe that the total number of edges, each with at least one vertex from $V'_{i+1} := V \setminus V_{i+1}$, is $m'_i := m - k_i - m_i - m_S - m_T$. Hence, the expected cut-size when vertices of V'_{i+1} are included, is at least

$$\max \begin{cases} k + m_T + m'_i/2 & \text{if } v_{i+1} \text{ is included in } S \\ k + m_S + m'_i/2 & \text{if } v_{i+1} \text{ is included in } T. \end{cases}$$

So, we choose the higher between them by simply putting v_{i+1} in S if $m_S < m_T$, or in T otherwise (Fig. 5.4). This is iterated from v_1 to v_n, as shown in Algorithm 17.

Algorithm 17: Derandomized algorithm for MAX-CUT$(G(V, E))$

```
1 S ← {v_1}, T ← ∅
2 for i ← 2, 3, ..., n do
3 |   m_S ← |{(u, v_i) : u ∈ S}|
4 |   m_T ← |{(u, v_i) : u ∈ T}|
5 |   if m_S < m_T then
6 |   |   S ← S ∪ {v_i}
7 |   else
8 |   |   T ← T ∪ {v_i}
9 return (S, T)
```

5.9 A Las Vegas Algorithm for BSP Tree

BSP tree has many applications in geometric algorithms and computer graphics. It provides spatial information about the objects in a scene, which helps rendering by ordering from front-to-back with respect to a viewer at a given location. It is also useful for ray tracing, geometrical operations with shapes, constructive solid geometry in CAD, collision detection in robotics, 3D video games, etc.

The figure aside shows a *sculpted bunny* made out of a block using around 9000 subtractions of dodecahedra. It is a recent research work that uses BSP for robustly performing Boolean operations on linear, 3D polyhedra [Gilbert Bernstein & Don Fussell: Fast, exact, linear Booleans. In *Proc. SGP 2009*, pp. 1269–1278]. The BSP-tree based system is 16-28 times faster at performing iterative computations than the best available system like CGAL's Nef Polyhedra-based system.

Binary space partitioning (BSP) recursively subdivides the space containing a set of physical objects—or even abstract objects, specially in higher dimension—using appropriate partition lines (2D) or partition planes (3D). The partitioned space, along with related objects or object parts, is efficiently represented by the BSP tree. It is a generalization of spatial tree structures like k-d trees and quadtrees. Unlike k-d trees and quadtrees where each partition line (or planes or hyperplanes) is always aligned along one of the coordinate axes (or coordinate planes), a partition line in BSP tree may have any orientation.

The algorithm for constructing a 2D BSP tree is recursive and quite simple. See the steps in Algorithm 24 and the demonstration in Fig. 5.5. For each line segment s_i, its left and right half-planes can easily be defined by treating s_i as a vector directed from its first endpoint to second endpoint. The size of the BSP tree varies with the permutation of the segments of S. We analyze here its expected size, i.e., the average size over all $n!$ permutations.

Algorithm 18: MakeBSP(S)

Input: $S = \{s_1, s_2, \ldots, s_n\}$ of n straight line segments in random order.
Output: A BSP tree for S.

if $n \leqslant 1$ **then**
  Create a tree T consisting of a single leaf node storing S.
  return T
else
  Define l_1 passing through s_1 as the partition line.
  Define l_1^- and l_1^+ as the respective left and right half-planes for l_1.
  $S^- \leftarrow \{s_i \in S \cap l_1^-\}$
  $T^- \leftarrow$ MakeBSP(S^-)
  $S^+ \leftarrow \{s_i \in S \cap l_1^+\}$
  $T^+ \leftarrow$ MakeBSP(S^+)
  Create a BSP tree T with root node corresponding to s_1 (and other segments, if any, that lie on l_1), left subtree T^-, right subtree T^+.
  return T.

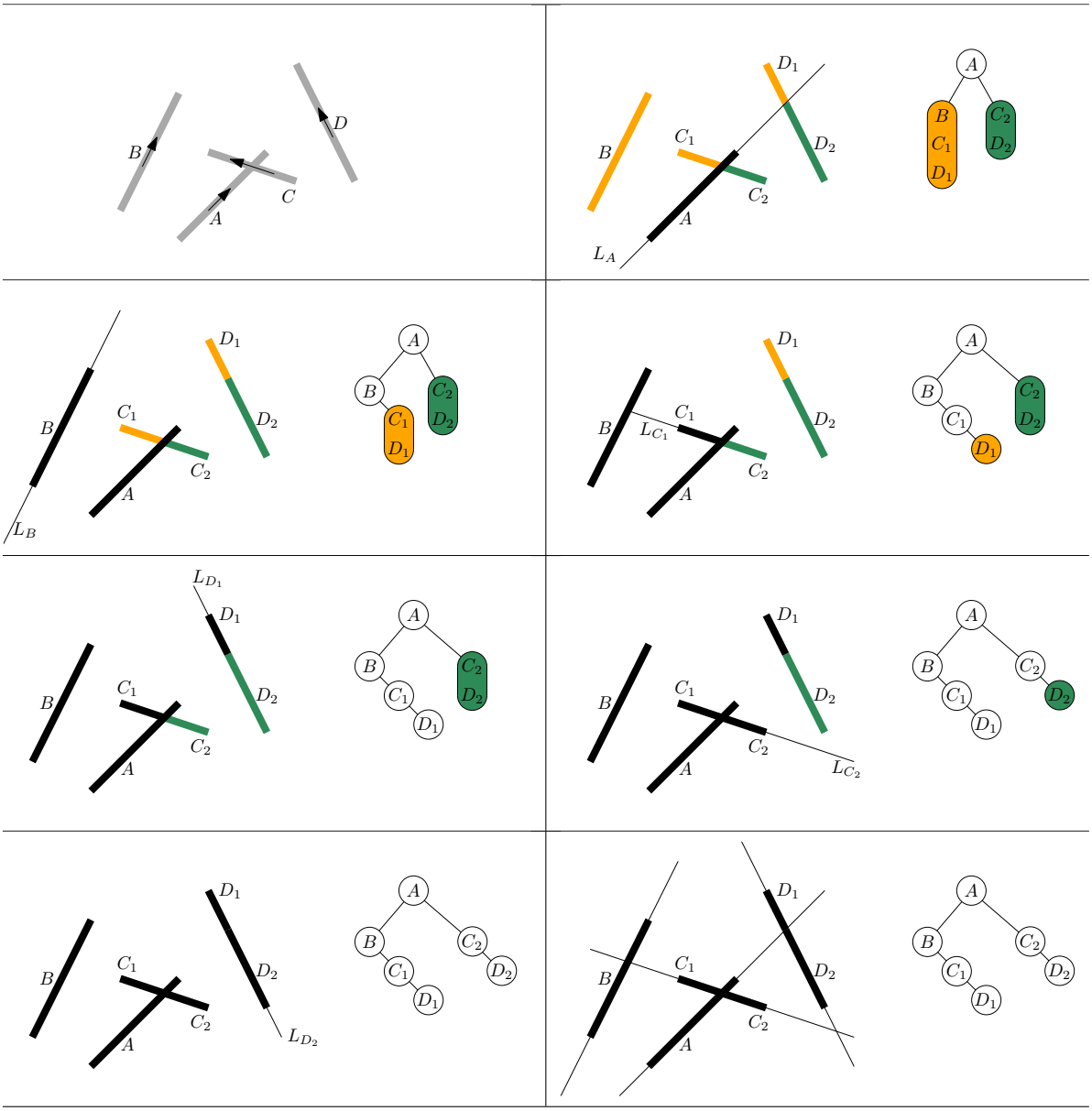


Figure 5.5: An example of randomized BSP tree formation for four line segments. Each segment is treated as a vector (shown by arrow) to distinguish its left and right half-planes.

When a particular segment s_i defines the partition line l_i, each other segment s_j may be intersected by l_i. If s_j gets split into two by l_i, then one segment of s_j will lie in the left half-plane, l_i^-, and the other in the right half-plane, l_i^+. If this happens for each $s_j \neq s_i$, then it would double the number of line segments after each partition; and the partitioning is unlikely to converge. Interestingly, it would not occur for the following reason.

Define $d(i, j)$ as the number of segments intersecting l_i in between s_i and s_j if s_j intersects l_i, and ∞ if s_j does not intersect l_i. For any finite non-negative integer c, there can be at most two segments for which $d(i, \cdot) = c$ (Fig. 5.6). Define $S(i, j)$ as the set containing s_i, s_j, and the segments lying between s_i and s_j with $d(i, \cdot) < d(i, j) < \infty$. If there exists a segment $s_k \in S(i, j)$, and the partition line l_k is selected before l_i, then the partition by l_i will not split

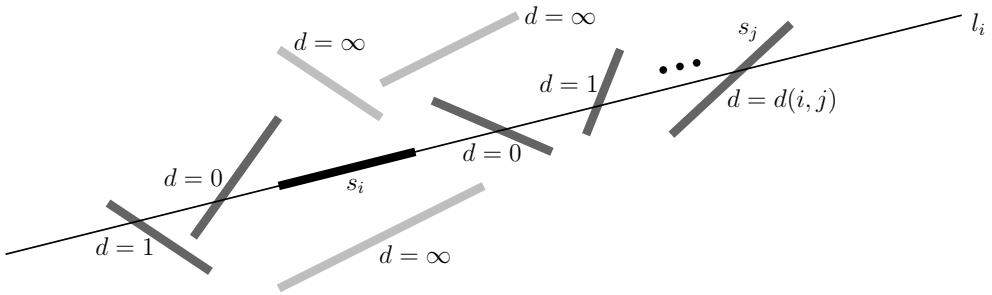


Figure 5.6: Distances of the line segments from s_i.

l_j, as l_j partitions the half-plane (defined by l_k) in which s_i does not lie; that is, s_k shields s_j in the sense that l_i cannot split s_j due to previously defined l_k. Hence, in order that l_i intersects s_j, all the segments of $S(i, j)$ should be selected only after s_j is selected (for defining their corresponding partition lines). So, the probability that l_i cuts s_j is simply the probability of occurrence of s_i as the first element in $S(i, j)$. As $S(i, j)$ contains $2 + d(i, j)$ elements, we have

$$\mathbf{P}[l_i \text{ cuts } s_j] \leqslant \frac{1}{d(i, j)},$$

the inequality sign being for the reason that there can be some segment(s) s_k with $d(i, k) = \infty$, but with l_k shielding s_j from l_i.

Hence, the expected number of cuts by l_i is at most

$$\sum_{j \neq i} \tfrac{1}{d(i,j)+2} \leqslant 2 \sum_{c=0}^{n-2} \tfrac{1}{c+2} \leqslant 2 \ln n.$$

By linearity of expectation, the expected number of cuts in total for all n segments is at most $2n \ln n$. As there are n segments to start with, the total number of segments after all cuts is no more than $n + 2 \ln n = O(n \log n)$.

Time complexity: As there are at most $O(n \log n)$ segments, the number of recursive calls of MakeBSP would be at most $O(n \log n)$. Each call is for n segments in the worst case, since there can be at most n segments (in parts or in full) in a particular region given by the intersection of its corresponding half-planes. Hence, the expected time complexity would be $O(n^2 \log n)$.

Chapter 6

NP-Completeness

Solving problems—especially the difficult ones like visiting all major cities in a country as quickly as possible—requires experience, introspection, diligence, as well as intelligence. Psychologists project the problem-solving faculty as the concluding part of a larger process that also includes *problem finding* and *problem shaping*. For solving apparently new problems, ideas from previously solved problems often come to use. A man who can easily connect to such previously solved problems is thought to be gifted with solving new problems.

An act of problem solving comprises a cycle in which we recognize the problem, define the problem, organize our domain knowledge, and then try to develop a strategy towards its solution. When the problem looks difficult, it involves proper abstraction and brainstorming to relate it to some analogous problem. The vertical logic of working out the solution step-by-step often fails in this case. Instead, a lateral thinking perhaps helps—in approaching solutions indirectly and creatively—for reasoning out, which is not immediately obvious. It rests on a horizontal imagination—flowing with with weird ideas—seemingly distancing itself from commonly followed vertical logic. And here comes the magic—the solver looks like a magician or a witch—as the beholders practically have no clue until the proof.

6.1 Introduction

We have many well-defined problems—some solved, some unsolved, and some yet to encounter. When the problem is easy, the solution comes fast. But when it is difficult, the solution cannot be obtained so quickly. The term "so quickly" is somewhat vague, since for a nursery kid, even addition or multiplication of two small numbers takes so much time! So, we need to fix the scientific meaning of what's an easy or what's a difficult problem. In other words, we have to classify the problems in general, and when a new problem is encountered, we need to find its class. The classification of a problem is done based on the type of of the best possible algorithm that can be used to solve it. Alternatively, we can also do such classification based on the already-known class of a similar problem. In that case, scientifically relating a new problem to a similar problem of known class is most important. To see how this can be done, we first require the following definitions.

Definition 6.1.1 (Polynomial-time algorithm) *The algorithm whose worst-case runtime on any input of size n is $O(n^k)$ for some constant k, is called a polynomial-time algorithm.*
Examples—Algorithms for sorting, searching, BFS, DFS, MST and shortest path finding for graphs, matrix multiplication, string matching, convex hull finding, etc.

Definition 6.1.2 (Non-polynomial algorithm) *If the runtime of an algorithm is superpolynomial or exponential in the input size, then it is a non-polynomial algorithm.*
Examples—Finding a clique of size k in an undirected graph $G(V, E)$; Coloring the vertices of a graph with minimum number of colors so that no two adjacent vertices get the same color.

A function that grows faster than any power of n is called ***superpolynomial***. One that grows slower than any exponential function of the form c^n is called ***subexponential***. An algorithm can require time that is both super-polynomial and sub-exponential; examples of this include the fastest known algorithms for ***integer factorization***.

Definition 6.1.3 (Deterministic algorithm) *An algorithm for which the output of every intermediate step is uniquely defined for a given input is called a deterministic algorithm.*
Examples—Quick sort, Merge sort, Heap sort, etc.; Binary search, Prim's algorithm for MST, etc.

Definition 6.1.4 (Nondeterministic algorithm) *If we allow an algorithm to contain certain operations whose outcome is not uniquely defined but is limited to a specified set of possibilities, then such an algorithm is called a* ***nondeterministic algorithm****. A machine executing such operations is allowed to choose any one of these outcomes subject to a terminating condition.*
Examples—Nondeterministic sorting.

Definition 6.1.5 (Decision problem) *A problem whose solution is binary (i.e., either '0' or '1') is called a* ***decision problem****. An algorithm for a decision problem is termed as a* ***decision algorithm****.*
Examples—Whether there a clique of size k in an undirected graph $G(V, E)$.

Definition 6.1.6 (Optimization problem) *A problem that involves the identification of an optimal (either minimum or maximum) value of a given function is known as an* ***optimization problem****. An algorithm to solve an optimization problem is termed as an* ***optimization algorithm****.*
Examples—Finding MST; Finding a maximum clique in an undirected graph $G(V, E)$.

We consider only decision problems for problem classification. Because, many optimization optimization problems can be recast into decision problems with the property that a decision problem can be solved in polynomial time if and only if the corresponding optimization problem can. In other cases, we can at least make the statement that if the decision problem cannot be solved in polynomial time, then the optimization problem cannot either. For example, we cannot solve by a deterministic polynomial-time algorithm whether there is a clique of size k in G (decision problem). So, we can neither find the maximum clique in G (optimization problem) by any deterministic polynomial-time algorithm.

6.1.1 Nondeterministic Algorithms

A non-deterministic algorithm consists of two parts:

- Guessing the solution, and here lies the 'non-determinism';
- Checking or verifying the guessed solution in polynomial time.

Three functions are essential for a nondeterministic algorithm:

- CHOICE(S) 'arbitrarily' chooses one element of an input set S. Alternatively, CHOICE$(1, n)$ 'arbitrarily' chooses one integer in the closed interval $[1, n]$.
- FAILURE signals an unsuccessful completion.
- SUCCESS signals a successful completion.

Example: Searching an element x in an unordered list $A[1..n]$.

Algorithm NON-DET-SEARCH(x, A, n)
1. $j \leftarrow$ CHOICE$(1, n)$
2. **if** $A[j] = x$
3. **print** j
4. SUCCESS
5. FAILURE

Since the computing times for CHOICE, FAILURE, and SUCCESS are taken to be $O(1)$, the algorithm NON-DET-SEARCH has $O(1)$ time complexity.

Example: Sorting n positive elements in a list $A[1..n]$.

Algorithm NON-DET-SORT(A, n)
1. **for** $i \leftarrow 1$ to n
2. $B[i] \leftarrow 0$
3. **for** $i \leftarrow 1$ to n
4. $j \leftarrow$ CHOICE$(1, n)$
5. **if** $B[j] \neq 0$
6. FAILURE
7. $B[j] \leftarrow A[i]$
8. **for** $i \leftarrow 1$ to $n - 1$
9. **if** $B[i] > B[i + 1]$
10. FAILURE
11. **print** B
12. SUCCESS

The algorithm NON-DET-SORT has $O(n)$ time complexity.

Note the decision version of sorting: Given a list A containing n distinct elements, is there a permutation of these n elements of A, say in some other list B, such that $B[i + 1] > B[i]$, for $i = 1, 2, \ldots, n - 1$?

6.2 $\mathcal{NP}$-*hard* and $\mathcal{NP}$-*complete* Problems

Definition 6.2.1 (Complexity Class $\mathcal{P}$) *$\mathcal{P}$ denotes the set of all decision problems solvable by deterministic algorithms in polynomial time.*

Definition 6.2.2 (Complexity Class $\mathcal{NP}$) *$\mathcal{NP}$ denotes set of all decision problems solvable by nondeterministic algorithms in polynomial time.*

Definition 6.2.3 (Complexity Class co-$\mathcal{NP}$) *co-$\mathcal{NP}$ means complement of $\mathcal{NP}$. It is the class of decision problems for which a* NO*-instance can be verified in polynomial time.*

An example of a problem in $\mathcal{NP}$ (not in $\mathcal{P}$) is the *subset sum problem*: Given a finite set of integers, is there a non-empty subset that sums to zero? An YES-instance can be easily verified in linear time. Its complementary problem is: Given a finite set of integers, does *every* non-empty subset have a nonzero sum? Since a NO-instance can be verified in linear time, this complementary problem is in co-$\mathcal{NP}$.

If a problem is in $\mathcal{P}$, then it is also in $\mathcal{NP}$ as we can verify the solution in polynomial time. Similarly, any problem in $\mathcal{P}$ is also in co-$\mathcal{NP}$. Notice that the definition of $\mathcal{NP}$ (and co-$\mathcal{NP}$) is not symmetric. Just because we can verify every YES answer quickly, we may not be able to check NO answers quickly, and vice versa. For example, there is no deterministic polynomial-time algorithm to check whether a boolean circuit is *never* satisfiable. But again, we believe that $\mathcal{NP} \neq$ co-$\mathcal{NP}$, although there is no proof till date.

Since any deterministic algorithm is a special case of a nondeterministic one, it is evident that $\mathcal{P} \subseteq \mathcal{NP}$. What we do not know, and what has become the most famous open problem in computer science, is whether $\mathcal{P} = \mathcal{NP}$ or $\mathcal{P} \neq \mathcal{NP}$. S. Cook formulated the following question: Is there any single problem X in $\mathcal{NP}$ such that, if we can show X to be in $\mathcal{P}$, then that would imply that $\mathcal{P}$=$\mathcal{NP}$? The following theorem by Cook provides an answer to this:

Theorem 6.2.1 (Cook's Theorem) *Satisfiability* (SAT) $\in \mathcal{P}$ *iff* $\mathcal{P} = \mathcal{NP}$.

Proof. See [12].

An immediate result of Cook's Theorem is ***Theory of* $\mathcal{NP}$-completeness**. It says, if any $\mathcal{NP}$*-complete* problem can be solved in polynomial time, then all $\mathcal{NP}$ problems can be solved in polynomial time ($\mathcal{P} = \mathcal{NP}$). Before discussing further on this, we have the following definitions.

Definition 6.2.4 (Reducibility) *A problem X reduces to another problem X' (in symbols, $X \leqslant_P X'$) if and only if each instance of X can be converted to some instance of X' in polynomial time by a deterministic algorithm (reduction algorithm).*

Example: A linear equation reduces to a quadratic equation.
Given an instance of $ax + b = 0$, we transform it to $0x^2 + ax + b = 0$, and solve the latter. Since the problem of solving a linear equation reduces to that of solving a quadratic equation, we say that solving a linear equation is no harder than solving a quadratic equation.

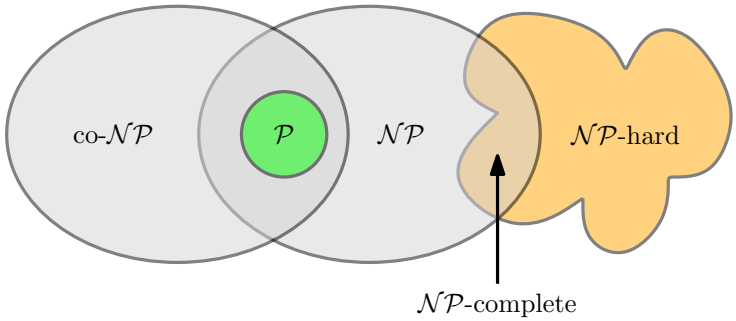


Figure 6.1: Relationship among some well-known problem classes.

Definition 6.2.5 (Polynomially Equivalent) *Two problems X and X' are polynomially equivalent if and only if $X \leqslant_P X'$ and $X' \leqslant_P X$.*

Definition 6.2.6 ($\mathcal{NP}$-*hard* problem) *A problem X is* $\mathcal{NP}$-hard *if and only if $X' \leqslant_P X$, $\forall\ X' \in \mathcal{NP}$. Alternatively, X is* $\mathcal{NP}$-hard *if and only if SAT $\leqslant_P X$.*

Alternative Definition: A problem is $\mathcal{NP}$-*hard* if an algorithm for solving it can be translated into one for solving any problem in $\mathcal{NP}$. An $\mathcal{NP}$-*hard* problem is, therefore, "at least as hard as any $\mathcal{NP}$-problem", although it might, in fact, be harder.

Definition 6.2.7 ($\mathcal{NP}$-*complete* problem) *A problem X is* $\mathcal{NP}$-complete *if and only if X is* $\mathcal{NP}$-hard *and $X \in \mathcal{NP}$.*

6.2.1 Proving a problem X to be $\mathcal{NP}$-*hard* or $\mathcal{NP}$-*complete*

1. Pick a problem X' (or SAT) already known to be $\mathcal{NP}$-*hard* or $\mathcal{NP}$-*complete*.
2. Propose a polynomial-time deterministic algorithm that can derive an instance I of X from *any instance* I' of X'. Thus, $X' \leqslant_P X$.
3. Since $\leqslant_P$ is a transitive relation, and X' is $\mathcal{NP}$-*hard* or $\mathcal{NP}$-*complete* (i.e., SAT $\leqslant_P X'$), we get SAT $\leqslant_P X$. Now conclude that X is $\mathcal{NP}$-*hard*.
4. Propose a nondeterministic polynomial-time algorithm for X to show that $X \in \mathcal{NP}$. This completes the proof that X is $\mathcal{NP}$-*complete*.

We have discussed about some well-known $\mathcal{NP}$-*complete* problems and have reasoned out why they are so. See Fig. 6.2 for a chain of reduction from one to the other. Before discussing all these, we show that there are problems which are even harder that $\mathcal{NP}$-*complete* problems. Halting problem, discussed in the following section, is such a problem. It is shown to be $\mathcal{NP}$-*hard*, as it is reducible from SAT, but found to be not in $\mathcal{NP}$.

6.2.2 Halting Problem

The halting problem is $\mathcal{NP}$-*hard* but not $\mathcal{NP}$-*complete*.

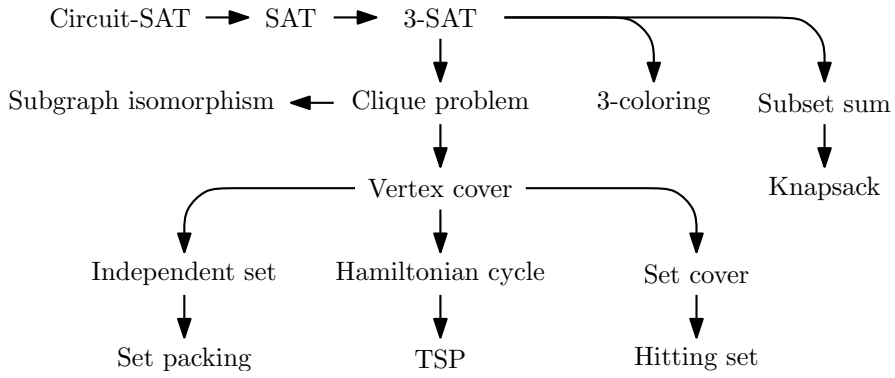


Figure 6.2: Reduction from one problem to another for a chain of common $\mathcal{NP}$-complete problems.

Proof. The halting problem is to determine, for an arbitrary deterministic algorithm A and an arbitrary input I, whether the algorithm A with input I ever terminates (or enters an infinite loop). Since there exists no algorithm (of any complexity) to solve this problem (see [12], page 506), it is not in $\mathcal{NP}$.

To show SAT $\leqslant_P$ HP, simply construct a deterministic algorithm A whose input is an arbitrary boolean formula X with n variables. Then A tries out all 2^n possible certificates and verifies whether X is satisfiable. If X is satisfiable, then A stops, and if not, then A enters an infinite loop. That is, A halts on some input if and only if X is satisfiable. Thus, if we had a polynomial-time algorithm H for the halting problem, then we could use the algorithm H to check (in polynomial time) whether or not the algorithm A halts on X, thereby solving the satisfiability problem — corresponding to algorithm A and input X — in polynomial time. Hence, the satisfiability problem reduces to halting problem.

6.3 Well-known $\mathcal{NP}$-*complete* Problems

Proofs of first few $\mathcal{NP}$-*complete* problems discussed here are available in [10]. So you should read those from [10]. For the rest, proofs or proof outlines are given in this section.

6.3.1 Satisfiability Problem (SAT)

To determine whether a boolean circuit is true for some assignment of truth values (input certificate) to the input binary variables. This problem is $\mathcal{NP}$-*complete*, since any problem in $\mathcal{NP}$ can be reduced to SAT in polynomial time. For its proof, see [10].

6.3.2 Formula Satisfiability Problem (ϕ-SAT)

To determine whether a boolean formula is true for some assignment of truth values (input certificate) to the input binary variables. This problem is $\mathcal{NP}$-*complete*, as it is in $\mathcal{NP}$, and it can be reduced from SAT in polynomial time [10].

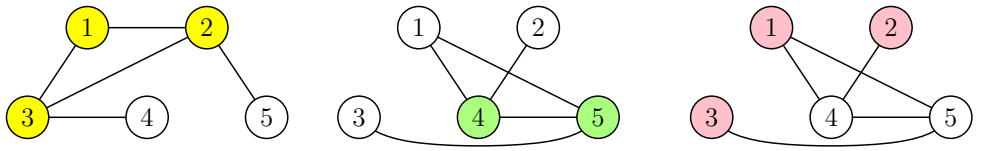


Figure 6.3: Clique K in $G \Leftrightarrow$ vertex cover $V \setminus K$ in $\bar{G} \Leftrightarrow$ independent set K in $\bar{G}$.

6.3.3 CNF-satisfiability

A CNF (conjunctive normal form) formula is represented in the conjunctive form $c_1 \wedge c_2 \ldots \wedge c_k$, where each clause c_i is represented as disjunction of one or more boolean variables or literals. Given a boolean formula ϕ in CNF, the problem of CNF-satisfiability is to decide whether ϕ is satisfiable. Any instance of this problem can be verified in polynomial time. And as it can be reduced from SAT, it is also $\mathcal{NP}$-*complete*; see [10] for its proof. If each clause in the CNF formula has exactly three literals, then the corresponding problem is called 3-SAT, which is also $\mathcal{NP}$-*complete*.

6.3.4 Max Clique Problem

A clique is a complete subgraph of a graph G. The size of a clique is the number of vertices in it. The max clique problem is an optimization problem that has to determine the size of a/the largest clique in G. This problem is $\mathcal{NP}$-*complete*, as its decision version—whether G has a clique of size at least k for some given k—is in $\mathcal{NP}$ and can be reduced from 3-SAT [10].

6.3.5 Vertex Cover

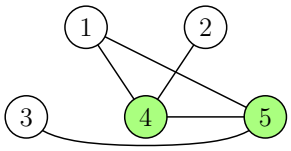
A vertex cover of an undirected graph $G(V, E)$ is a subset $V' \subseteq V$ such that each edge of E has at least one vertex in V'. The vertex cover V' thus covers all the edges of G. The size of the vertex cover is given by the number of vertices in V'. The decision problem is to determine, for a given positive integer k, whether G has a vertex cover of size k. This problem is in $\mathcal{NP}$, as it can be verified in polynomial time. We show here that the problem is $\mathcal{NP}$-*complete* by reducing it from the clique decision problem.

Construct the complement of G as $\bar{G}(V, \bar{E})$, where $\bar{E} = V^2 \setminus E$. We now prove that G has a k-clique if and only if $\bar{G}$ has a vertex cover of size $|V| - k$. In particular, we prove that $K \subseteq V$ is a clique in G if and only if $V' := V \setminus K$ is a vertex cover in $\bar{G}$ (Fig. 6.3).

Consider any edge $(u, v) \in \bar{E}$. Clearly, $(u, v) \notin E$, which implies that at least one vertex between u and v, say u w.l.o.g., is not in K; so, $u \in V'$ and it covers (u, v). Hence, V' is a vertex cover.

Conversely, let $V' \subseteq V$ be a vertex cover in $\bar{G}$. If $(u, v) \in \bar{E}$, then at least one vertex between u and v is in V'. This, in turn, implies that for any $(u, v) \in V^2$, if $u \notin V'$ and $v \notin V'$, then $(u, v) \notin \bar{E}$, which means $(u, v) \in E$ with $u \in K$ and $v \in K$. So, K is a clique.

It is easy to see now that a max clique in G implies the max value of k, and so gives a min vertex cover in $\bar{G}$.



vertices ≡ subsets of U	edges ≡ elements of U				
	(1,4)	(1,5)	(2,4)	(3,5)	(4,5)
1	1	1			
2			1		
3				1	
4	1		1		1
5		1		1	1

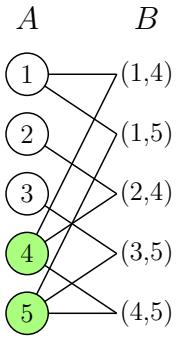


Figure 6.4: **Left:** Vertex cover $V' = \{4, 5\}$ in $G \Leftrightarrow$ **Middle:** Set cover $S = \{S_4, S_5\}$. **Right:** Hitting set $\{4, 5\}$ that covers $B \equiv \mathcal{F}$. Each element of B is an edge, which is conceived as a two-element set.

Note: The vertex cover problem is often useful to prove the $\mathcal{NP}$-completeness of other problems. Figure 6.2 shows some of these that are reduced from it.

6.3.6 Independent Set

An independent set of an undirected graph $G(V, E)$ is a subset $S \subseteq V$ such that there does not exist any edge of E between any two vertices of S. The size of the independent set is given by the number of vertices in S. The decision problem is to determine, for a given positive integer s, whether G has an independent set of size s. This problem is in $\mathcal{NP}$, as it can be verified in polynomial time. It is easy to show that the problem is $\mathcal{NP}$*-complete* by reducing it from the vertex cover problem (Fig. 6.3). The proof is based on the fact that for any vertex cover V' of G, $\{u, v\} \cap V' = \emptyset \Leftrightarrow (u, v) \notin E \Leftrightarrow \{u, v\} \subseteq S$.

6.3.7 Set Cover

Given a set U of n elements, a family $\mathcal{F} = \{S_1, \ldots, S_m\}$ of subsets of U, and a number k, the decision problem is to determine whether there exists a subfamily (i.e., set cover) with at most k of these subsets whose union is U. This problem is in $\mathcal{NP}$, as it can be verified in polynomial time. We show that the problem is $\mathcal{NP}$*-complete* by reducing it from the vertex cover problem. Given $G(V, E)$ and its vertex cover V', we prepare U and $\mathcal{F}$, and derive a set cover $\mathcal{F}'$, as shown in Fig. 6.4. Each edge in E has 1-to-1 correspondence with a unique element in U. Each vertex v_i in V has 1-to-1 correspondence with a unique subset S_i in $\mathcal{F}$, and the elements in S_i are in 1-to-1 correspondence with the vertices adjacent to v_i. Now we prepare $\mathcal{F}'$ from V', using the vertex-element correspondence between V and $\mathcal{F}$.

Each edge $(u, v) \in E$ is covered by at least one of its vertices in V'. Let, w.l.o.g., u be that vertex. Then the element corresponding to (u, v) in U belongs to the set in $\mathcal{F}'$ corresponding to u. Hence, all the elements of U are covered by $\mathcal{F}'$. The converse proof is similar and easy to follow.

6.3.8 Hitting Set

Given a set U of n elements, a family $\mathcal{F} = \{S_1, \ldots, S_m\}$ of subsets of U, and a number k, we have to determine whether there exists a subset (hitting set) S of U with at most k elements, such that $S_i \cap S \neq \emptyset$ for each $S_i \in \mathcal{F}$.

The hitting set problem can be reduced from the set cover problem. A sketch of the proof is as follows. An instance of set cover can be viewed as a bipartite graph $G(V, E)$, $V = A \cup B$, with $A \equiv \mathcal{F}$, $B \equiv U$, and E representing the inclusion of elements in sets of $\mathcal{F}$, as illustrated in Fig. 6.4. The task is then to find a subset of A having k vertices, which covers all vertices of B. In the hitting set problem, the objective is to cover the vertices of A using a subset of B with a given cardinality. On interchanging A and B, the problem is solved. Just make $B \equiv \mathcal{F}$ and $A \equiv U$, and then the hitting set is a subset of A having k vertices and covering all vertices of B.

6.3.9 Set Packing

Given a set U of n elements, a family $\mathcal{F} = \{S_1, \ldots, S_m\}$ of subsets of U, and a number k, the decision problem is to determine whether there exists a subfamily (i.e., packing) with at most k of these subsets, which are pairwise disjoint. This problem is in $\mathcal{NP}$, as it can be verified in polynomial time. The reduction ideas of Sec. 6.3.6 and Sec. 6.3.7 can be used to show that this problem is also $\mathcal{NP}$-*complete* by reducing it from Independent Set problem.

6.3.10 Subset Sum

It is a decision problem. Given a set S of n natural numbers and a target number t, the problem is to decide whether there exists a subset of S whose elements add up to t. As any instance can be verified in poly-time, the problem is in $\mathcal{NP}$. It can be reduced from vertex cover problem, and hence it is $\mathcal{NP}$-*complete*. See [10] for the reduction.

6.3.11 0-1 Knapsack

In Sec. 2.3, we have seen a pseudo-polynomial algorithm for optimization version of 0-1 knapsack problem. The decision version is as follows. We are given a set $A = \{a_i : i = 1, 2, \ldots, n\}$ whose each item a_i has a benefit b_i and a positive weight w_i. We are given a total benefit b and a knapsack whose weight carrying capacity is k. The problem is to decide whether we can select items from A such that their total weight is at most k and total benefit is at least b. Verification of this problem needs polynomial time, and so it is in $\mathcal{NP}$. It is $\mathcal{NP}$-*complete*, since we can reduce it from the subset sum problem in poly time. Let (S, t) be an instance of the subset sum problem, where $S = \{s_1, s_2, \ldots, s_n\}$. Now, the reduction steps are as follows:
Prepare A with $w_i = b_i = s_i$ for $i = 1, 2, \ldots, n$.
Make $k = b = t$.
Now, we can easily show that the answer to (S, t) is YES if and only if there is a subset of A whose elements have total weight at most $k(= t)$ and total benefit at least $b(= t)$.

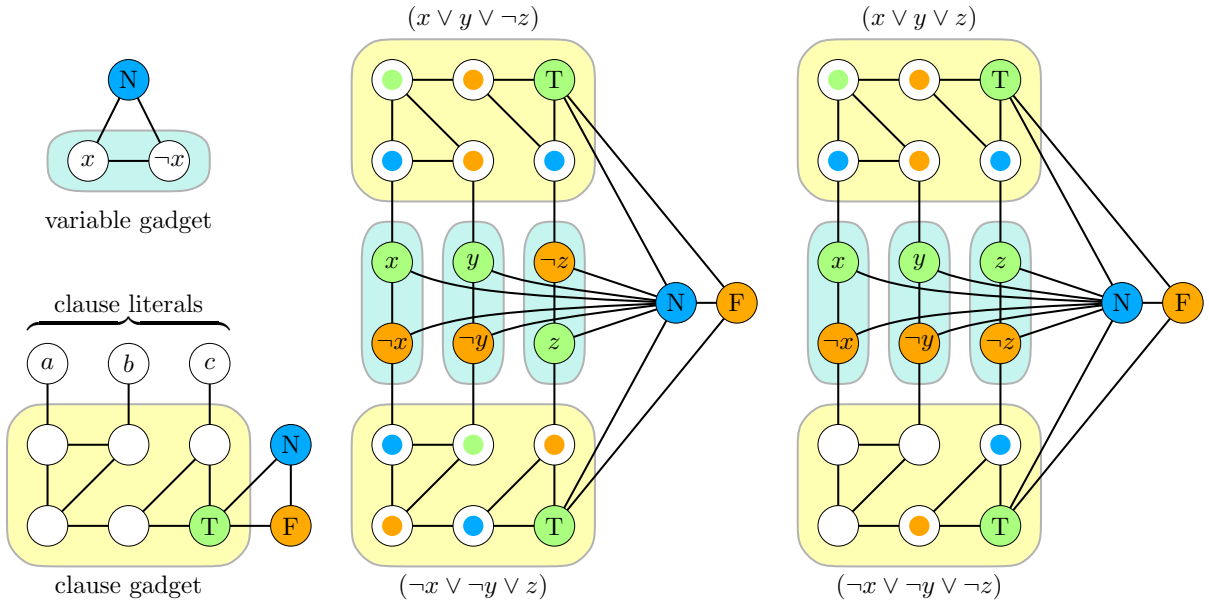


Figure 6.5: Two examples of reduction of 3-CNF formulas. **Left:** Two types of gadgets used for reduction. **Middle:** $\phi_1 = (x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z)$ is satisfiable for $\langle 1, 1, 1\rangle \Rightarrow G$ is 3-colorable. **Right:** $\phi_2 = (x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$ is not satisfiable for $\langle 1, 1, 1\rangle \Rightarrow G$ is not 3-colorable for this $\langle 1, 1, 1\rangle$. The reason is that if the second clause is 3-colored, then out of its two nodes adjacent to to $\neg x$ and $\neg y$, one has to be FALSE, which violates 3-coloring. However, for a different input like $\langle 1, 1, 0\rangle$, $\phi_2 =$ TRUE, and then G is 3-colorable.

6.3.12 3-Coloring

Given an undirected graph $G(V, E)$, we have to decide whether its vertices can be colored by 3 colors so that no two adjacent vertices get the same color. This problem can be shown to be *$\mathcal{NP}$-complete* by a reduction from 3-SAT. The reduction uses two special widgets or gadgets (*variable gadget* and *clause gadget*) and two special graph vertices, labeled as N (NEUTRAL) and F (FALSE), that are not part of any gadget.

As shown in Fig. 6.5, a gadget variable consists of two vertices, x and $\neg x$, connected in a triangle with the NEUTRAL vertex. The gadget for a clause $(a \vee b \vee c)$ consists of six vertices, connected in a purposeful manner to the vertices representing the literals a, b, c, and to the NEUTRAL and the FALSE vertices. A 3-CNF formula ϕ is converted into an undirected graph by using a variable gadget for each of its variables and a clause gadget for each of its clauses, and connecting them, as shown.

In any 3-coloring of G, we designate the three colors as TRUE, FALSE, and NEUTRAL. A variable gadget can have two possible colorings: the vertex labeled with the variable x is colored either TRUE or FALSE, and the vertex labeled with $\neg x$ is correspondingly colored either FALSE or TRUE. This ensures that valid coloring to the variable gadgets is in 1-to-1 correspondence with truth assignments to the variables, and hence the coloring of the variable gadget simulates the behavior of a variable's truth assignment.

Each clause assignment has a valid 3-coloring if at least one of its adjacent term vertices is colored TRUE, and cannot be 3-colored if all of its adjacent term vertices are colored FALSE. This ensures that the clause gadget can be colored if and only if the corresponding truth assignment satisfies

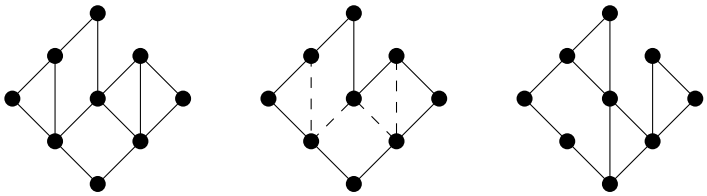


Figure 6.6: Presence or absence of Hamiltonian cycles in graphs.

the clause, and hence the coloring of the clause gadget also simulates the behavior of a clause.

The gadgets and their coloring, therefore, ensures that a 3-CNF formula ϕ is satisfiable if and only if the corresponding graph G is 3-colorable.

6.3.13 Hamiltonian Cycle

A ***Hamiltonian path*** in an undirected or a directed graph G is a path that visits each vertex exactly once. When the path is a cycle, it is called a ***Hamiltonian cycle*** (or Hamiltonian circuit). See Fig. 6.7 for examples. Any Hamiltonian cycle can be converted to a Hamiltonian path by removing one of its edges, but a Hamiltonian path can be extended to Hamiltonian cycle only if the first and the last vertices of the path are adjacent in G. Determining whether a Hamiltonian path or a Hamiltonian cycle exists in a graph G is $\mathcal{NP}$-*complete*. The Ham-cycle problem can be reduced from 3-SAT using certain widgets, see [10].

6.3.14 Traveling Salesman Problem

Given a set of cities and the distance of each from each other, the ***traveling salesman problem*** (TSP) is to find the shortest route that visits each city exactly once and returns to the start city. TSP can be modeled as finding the minimum-weight cycle comprising all vertices in an undirected weighted complete graph $G(V, E)$, such that V corresponds to the set of cities, $E = V \times V$, and weight of an edge in E is the distance between two corresponding cities.

The decision version of the problem is to find a cycle of a given weight in G. TSP is in $\mathcal{NP}$, as an instance of TSP can easily be verified in polynomial time. It can be reduced from Ham-cycle problem, and hence it is $\mathcal{NP}$-*complete*. The proof is as follows. Let $G(V, E)$ be an instance of the Ham-cycle problem. Construct $G'(V, E')$ from G, with $E' = V' \times V'$. For each $e \in E$, assign weight of e as 0 in E'; for each other edge in E', assign weight as 1. Now it is easy to prove that G has a Ham-cycle if and only if G' contains a TSP path of total weight 0.

6.3.15 Subgraph Isomorphism Problem

Two graphs are said to be ***isomorphic*** if we can relabel the vertices of one so that it becomes identical with the other. Formally, two graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ are isomorphic to each other if there exists a function $f : V_1 \mapsto V_2$ such that $(f(u), f(v)) \in E_2$ if and only if $(u, v) \in E_1$. In simpler words, $f(G_1) \equiv G_2$ if and only if G_1 and G_2 are isomorphic.

Given two graphs G_1 and G_2, the ***subgraph isomorphism problem*** is to determine whether there exists a subgraph G_1' in G_1 so that G_1' is isomorphic to G_2. Given an instance of the

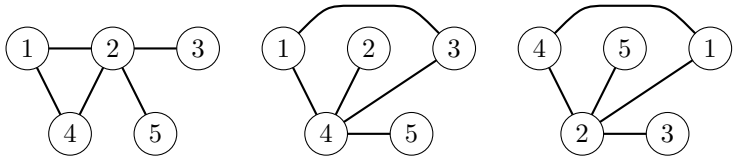


Figure 6.7: An example of isomorphism. $f(G_1) \equiv G_2$ with $f(v_i) = ((i+2) \bmod 5) + 1$ for each $v_i \in V_1$.

reassigned vertex labels of G_1, the verification whether $G_2(V_2, E_2)$ has $E_2 = E_1 \cap V_{1k} \times V_{1k}$, where $k = |V_2|$, can easily be done in polynomial time. Hence, it is in $\mathcal{NP}$. It can be further shown to be $\mathcal{NP}$-*complete*, by reducing it from the clique decision problem. Let (G_1, k) be an instance of the clique decision problem. Consider $G_2(V_2, E_2)$ as the complete graph with $|V_2| = k$ and $E_2 = V_2 \times V_2$. Then it is easy to prove that (G_1, G_2) is an instance of subgraph isomorphism problem with the property that there is a k-clique in G_1 if and only if G_2 is isomorphic to a subgraph in G_1.

Concluding notes

The concept of $\mathcal{NP}$-completeness was developed in 1960s–70s in the US and the USSR. In the US in 1971, Stephen Cook published his paper: *The complexity of theorem proving procedures* [7]. Richard Karp's subsequent paper, titled *Reducibility among combinatorial problems* [13], provided a list of 21 $\mathcal{NP}$-*complete* problems[1] and generated further interests among computer scientists. Cook and Karp received Turing Award for this work.

[1] See http://en.wikipedia.org/wiki/Karp%27s_21_NP-complete_problems.

Chapter 7

Approximation Algorithms

7.1 Introduction

Approximation algorithms are used to find approximate or sub-optimal solutions to optimization problems, especially when the problems are difficult in nature, such as $\mathcal{NP}$-*hard* or $\mathcal{NP}$-*complete* problems. Opposed to heuristics, an approximation algorithm, in principle, not only finds an approximate solution reasonably fast, but also guarantees quality of the solution with provable bounds on low-order polynomial runtime.

Approximation algorithms for $\mathcal{NP}$-*complete* problems vary greatly in their extent of approximation. Some can approximate within a constant factor and they are in polynomial-time approximation scheme or PTAS, e.g., bin packing problem or vertex cover problem. Others cannot approximate within any constant, or even a polynomial factor, unless $\mathcal{P} = \mathcal{NP}$, e.g., maximum clique problem.

Approximation algorithms can be designed for optimization problems only, and not for decision problems like SAT, although it is often possible to conceive optimization versions of such problems, such as the maximum satisfiability problem (Max-SAT).

7.2 Performance Measure

Let P be a minimization or a maximization problem. Let $\varepsilon > 0$. Let $\rho = 1 + \varepsilon$ for minimization and $\rho = 1 - \varepsilon$ for maximization, as the case may be. An algorithm A is called a ***ρ-approximation algorithm*** for problem P, if for every instance of P, it delivers a feasible solution with objective value X, such that

$$|X - X^*| \leqslant \varepsilon \cdot X^*.$$

In this case, the value ρ is called the ***performance guarantee*** or the ***worst-case ratio*** of the approximation algorithm A. Note that in case of minimization, the above inequality becomes $X \leqslant (1 + \varepsilon)X^*$, whereas for maximization, it becomes $X \geqslant (1 - \varepsilon)X^*$. Hence, for minimization problems, the worst-case ratio is $\rho = 1 + \varepsilon > 1$, and for maximization problems, $0 < \rho = 1 - \varepsilon \leqslant 1$. The value ρ can be viewed as the quality measure of the approximation algorithm. The closer ρ is to 1, the lower the value of ε, and the better is the algorithm.

We have different complexity classes of problems based on the nature of their approximate solutions. The complexity class APX comprises all minimization problems that have polynomial-time approximation algorithms with some finite worst-case ratio, and all maximization problems that have polynomial-time approximation algorithms with some positive worst-case ratio. PTAS and FPTAS are two sub-classes of APX (in fact, PTAS contains FPTAS). PTAS means the

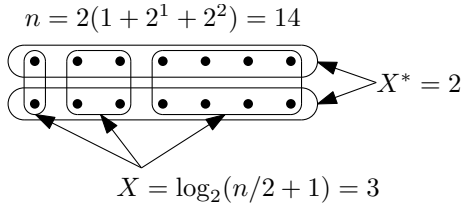


Figure 7.1: Worst-case instance of the greedy approximation algorithm for minimum set cover. Here, $U = \{s_1, \ldots, s_{14}\}$ has $n = 14$ elements. First selected is $S_1 = \{s_1, \ldots, s_8\}$, then $S_2 = \{s_9, \ldots, s_{12}\}$, and then $S_3 = \{s_{13}, s_{14}\}$, which gives $X = 3$. The optimal solution is for $S_3 = \{s_i : i \mod 2 = 0\}$ and $S_4 = \{s_i : i \mod 2 = 1\}$.

class of problems having polynomial-time approximation schemes in the sense that their time complexity is polynomial in the input size (e.g., Euclidean TSP). FPTAS means the class of all fully polynomial-time approximation schemes, i.e., time complexity is polynomial in both input size and $1/\varepsilon$. Following is the commonly believed relation among the above classes.

$$\boxed{\mathcal{NP} \supset \text{APX} \supset \text{PTAS} \supset \text{FPTAS} \supset \mathcal{P}}.$$

In PTAS or FPTAS, an algorithm takes an instance of the optimization problem and a positive value of the parameter ε, and in polynomial time, produces a solution that is within a factor ρ of optimal. For example, for the Euclidean TSP, a PTAS would produce a tour of length at most $\rho = (1 + \varepsilon)L$, L being the length of the shortest tour. The running time of a PTAS is required to be polynomial in n for every fixed ε but can be different for different ε. Thus, an algorithm with runtime $O(n^{1/\varepsilon})$ or even $O(n^{\exp(1/\varepsilon)})$ counts as a PTAS.

7.3 Minimum Set Cover

See the decision version of the problem in Sec. 6.3.7. The optimization version is: Given a set U of n elements and a family $\mathcal{F} = \{S_1, \ldots, S_m\}$ of subsets of U, find a/the smallest subfamily whose union is U. As the decision problem is $\mathcal{NP}$*-complete*, so also the optimization one. We have an approximation algorithm as follows. Chooses sets from $\mathcal{F}$ greedily: at each stage, choose the set that contains the largest number of uncovered elements. As shown in Fig. 7.1, there can be an input instance for which the algorithm outputs a cover size of $O(\log n)$, although the optimal solution is just $X^* = 2$. Hence, for a generalized instance extending from the above instance, the approximation ratio turns out to be $O(\log n)$, and so it is not a constant-factor approximation algorithm.
So a caution: Beware of the greedy strategy while designing an approximation algorithm.

7.4 Minimum Vertex Cover

Consider the following greedy algorithm to obtain the approximate minimum vertex cover V' of an undirected graph $G(V, E)$. Find a vertex v with maximum degree. Add v to V', and remove v and all its incident edges from G. Repeat until all the edges of E are covered.

How good is the above approximation algorithm? See the example in Fig. 7.2(left). Here, $X = 11$, as the algorithm is unlucky in breaking ties. The algorithm first chooses all the

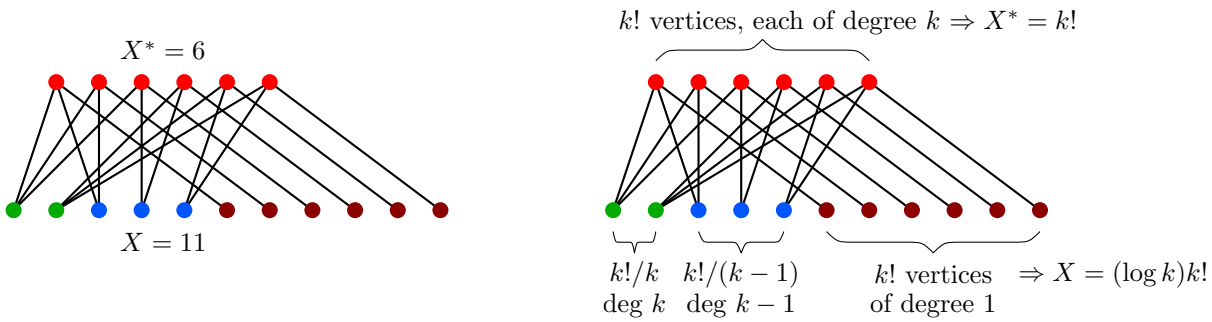


Figure 7.2: **Left:** A graph with minimum vertex cover size = 6 and approximate vertex cover size = 11. **Right:** A generalized example of the graph that shows $\rho = \log k$.

green vertices (degree 3), then all the blue vertices (degree 2), and then all the brown vertices (degree 1). Had it chosen all the red vertices (degree 3) one by one, it could get $X^* = 6$. The actual scenario is more explicit for the generalized example shown in Fig. 7.2(right). Hence, it is not a constant-factor approximation, since the approximation ratio is at least $\log k$, where $n = O((\log k)k!)$.
The lesson is: Greedy strategy seldom yields a good approximation algorithm.

Let's now look into the following algorithm. Choose any arbitrary edge (u, v). Include both u and v in V'. Remove u and v, and all the edges covered by them from G. Repeat this until there is no edge. It is a 2-approximation algorithm (i.e., $\rho = 2$), since at least one of u and v must be present in the minimum vertex cover for each edge $(u, v) \in E$.

7.5 Traveling Salesman Problem

As shown in Sec. 6.3.14, TSP is $\mathcal{NP}$-*complete*. Interestingly, there exists no polynomial-time constant-factor approximation algorithm for TSP unless $\mathcal{P} = \mathcal{NP}$. We prove it by contradiction. Let ρ be the constant approximation ratio, if possible, for an approximation algorithm of TSP. Let the undirected graph $G(V, E)$ be an instance of Hamiltonian cycle problem (Sec. 6.3.13). Let $G'(V, E' = V \times V, w)$ be the complete weighted graph with $w(u, v) = 1$ if $(u, v) \in E$, and $w(u, v) = \rho|V| + 1$ otherwise. Now, it is easy to see that if there is a Ham-cycle in G, then its corresponding tour in G' has a cost $X^* = |V|$, and so the approximation algorithm would return $X \leqslant \rho|V|$. And if there is no Ham-cycle in G, then any tour in G' has at least one edge with weight $\rho|V| + 1$. Let there be k such edges, each with weight $\rho|V| + 1$. Then $X^* = (|V| - k) \times 1 + k \times (\rho|V| + 1) = (1 + k\rho)|V| \geqslant (1 + \rho)|V|$ as $k \geqslant 1$, which implies $X \geqslant X^* > \rho|V|$. Hence, the value $\rho|V|$ makes all the difference—we can say, there is a Ham-cycle in G if and only if the approximation algorithm returns a cost not exceeding $\rho|V|$. If the approximation algorithm has polynomial time complexity, then the Ham-cycle problem is also solved in polynomial time, which implies $\mathcal{P} = \mathcal{NP}$—a contradiction.

7.6 Euclidean Traveling Salesman Problem

When the vertices of a complete, weighted, undirected graph $G(V, E, w)$ are embedded on the plane so that the weight of an edge $e(u, v)$ in E is given by the Euclidean distance between u and

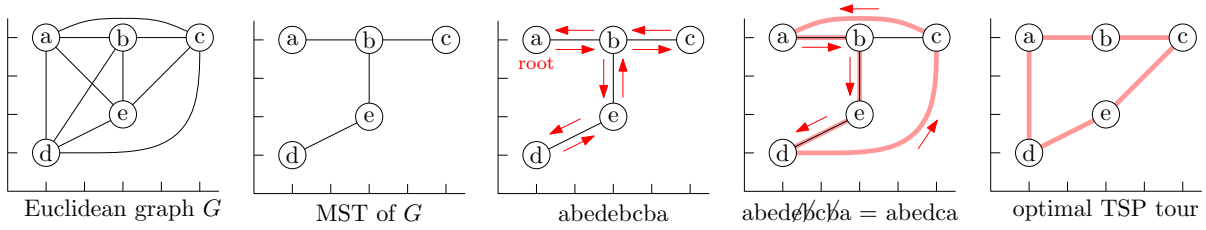


Figure 7.3: Approximation algorithm of TSP: $X = 2 + 2 + \sqrt{5} + 5 + 4 = 13 + \sqrt{5} \approx 15.24$, $X^* = 3 + \sqrt{5} + 2\sqrt{2} + 2 + 2 \approx 12.06$.

v, the shortest TSP tour in G can be solved by a polynomial-time constant-factor approximation algorithm as follows. Find the MST T of G. During the preorder travseral of T, remove the vertices that have appeared earlier, to get a tour covering all vertices of V. This gives an approximate solution of TSP. See, for example, Fig. 7.3.

In the preorder traversal, each edge of T occurs twice. During the preorder traversal of T, when we remove vertices that have appeared earlier, two or more edges of E are removed by a single edge of E, and due to *triangle inequality* in the Euclidean space, the length of the replacee edge cannot be greater than the sum of the lengths of the replaced edges. For example, in Fig. 7.3, the length of the edge dc is smaller than the sum of lengths of the replaced edges de, eb, bc. Hence, $X \leqslant 2T^*$,

Now, let X^* be the cost of the optimal TSP tour in G. Observe that removal of any edge in the optimal tour gives a spanning tree with cost less than X^*, which cannot be less than the cost T^* of T, as T is an MST. Hence, $T^* < X^*$.

From above two observations, we get $X < 2X^*$. Hence, the algorithm has approximation ratio $\rho < 2$.

Bibliography

[1] I. J. Balaban (1995), An optimal algorithm for finding segments intersections, in *Proc. 11th ACM Symp. Computational Geometry*, pp. 211–219.

[2] U. Bartuschka, K. Mehlhorn, and S. Näher (1997), A robust and efficient implementation of a sweep line algorithm for the straight line segment intersection problem, in *Proc. Workshop on Algorithm Engineering*, pp. 124–135.

[3] J. L. Bentley, and T. A. Ottmann (1979), Algorithms for reporting and counting geometric intersections, *IEEE Transactions on Computers*: C-28(9), pp. 643–647.

[4] B. Chazelle and H. Edelsbrunner (1992), An optimal algorithm for intersecting line segments in the plane, *Journal of the ACM*: 39(1): 1–54.

[5] K. L. Clarkson (1988), Applications of random sampling in computational geometry, in *Proc. 4th ACM Symp. Computational Geometry*, pp. 1–11.

[6] K. L. Clarkson, R. Cole, and R. E. Tarjan (1992), Randomized parallel algorithms for trapezoidal diagrams, *International Journal of Computational Geometry and Applications* 2(2): 117–133, Corrigendum 2(3): 341–343.

[7] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of Computing*, pages 151–158. ACM, 1971.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. Prentice Hall of India, New-Delhi, 2000.

[9] E. Horowitz, S. Sahni, and S. Rajasekaran. *Computer Algorithms*. Galgotia Publications Pvt. Ltd., New Delhi, India, 1999.

[10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. Prentice Hall of India, New-Delhi, 2000.

[11] D. Eppstein, M. Goodrich, and D. Strash (2009), Linear-time algorithms for geometric graphs with sublinearly many crossings, in *Proc. 20th ACM-SIAM Symp. Discrete Algorithms (SODA 2009)*, pp. 150–159.

[12] E. Horowitz, S. Sahni, and S. Rajasekaran. *Computer Algorithms*. Galgotia Publications Pvt. Ltd., New Delhi, India, 1999.

[13] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[14] K. Mulmuley (1988), A fast planar partition algorithm, in *Proc. 29th IEEE Symp. Foundations of Computer Science (FOCS 1988)*, pp. 580–589.

[15] F. P. Preparata and M. I. Shamos (1985), *Computational Geometry: An Introduction* (Theorem 7.6, p. 280 and Section 7.2.3: Intersection of line segments, pp. 278–287), Springer-Verlag.