KALINGA UNIVERSITY, NAYA RAIPUR

Faculty of Computer Science & IT

Programme- Bachelor of Computer Applications (AIML)

Course Name- Design and Analysis of Algorithm

Course Code- BCAAIML504 **5th Semester**

Unit-4

ITERATIVE IMPROVEMENT

It starts with some feasible solution (a solution that satisfies all the constraints of the problem) and proceeds to improve it by repeated applications of some simple step. This step typically involves a small, localized change yielding a feasible solution with an improved value of the objective function. When no such change improves the value of the objective function, the algorithm returns the last feasible solution as optimal and stops.

There can be several obstacles to the successful implementation of this idea. First, we need an initial feasible solution. Second, it is not always clear what changes should be allowed in a feasible solution so that we can check efficiently whether the current solution is locally optimal and, if not, replace it with a better one. Third—and this is the most fundamental difficulty— is an issue of local versus global extremum (maximum or minimum).

SIMPLEX METHOD

Converting a linear program to Standard Form

- Before the simplex algorithm can be applied, the linear program must be converted into standard form where all the constraints are written as equations (no inequalities) and all variables are nonnegative (no unrestricted variables).
- This process of converting a linear program to its standard form requires the addition of slack variable si which represents the amount of the resource not used in the ith constraint. Similarly, constraints can be converted into standard form by subtracting excess variable e_i .

- The standard form of any linear program can then be represented by the following linear system with n variables (including decision, slack and excess variables) and m constraints.

$$\max z$$

$$(or \min)$$

$$s.t. \quad \begin{array}{llllll} a_{11}x_1 + & a_{12}x_2 + & \cdots + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 + & a_{22}x_2 + & \cdots + & a_{2n}x_n & = & b_2 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \end{array}$$

$$a_{m1}\,x_1 \square a_{m2} \quad x_2 \ldots \square \begin{matrix} a_{mn} \\ x_n \end{matrix} \quad \begin{matrix} = \\ b_m \end{matrix}$$

EXAMPLE 1 Consider the following linear programming problem in two variables: maximize $3x + 5y$

subject to $x + y \le 4$

$x + 3y \le 6$

$x \ge 0,\ y \ge 0.$

By definition, a ***feasible solution*** to this problem is any point *(x, y)* that satisfies all the constraints of the problem; the problem's ***feasible region*** is the set of all its feasible points. Our task is to find an ***optimal solution***, a point in the feasible region with the largest value of the ***objective function*** $z = 3x + 5y$. Linear programming problems with the empty feasible region are called ***infeasible***. Obviously, infeasible problems do not have optimal solutions.

To begin with, before we can apply the simplex method to a linear programming problem, it has to be represented in a special form called the ***standard form***. The standard form has the following requirements:

- It must be a maximization problem.
- All the constraints (except the nonnegativity constraints) must be in the form of linear equations with nonnegative right-hand sides.
- All the variables must be required to be nonnegative.

Thus, the general linear programming problem in standard form with m constraints and n unknowns $(n \ge m)$ is

maximize $c1x1 + \ldots + cnxn$

subject to $ai1x1 + \ldots + ainxn = bi,$ where $bi \ge 0$ for $i = 1, 2, \ldots, m$

$x1 \ge 0, \ldots, xn \ge 0.$

It can also be written in compact matrix notations:

maximize cx

subject to $Ax = b$

$x \ge 0,$ where

$$c = [c_1\ c_2\ \dots\ c_n],\quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix},\quad A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix},\quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}.$$

Any linear programming problem can be transformed into an equivalent problem in standard form. If an objective function needs to be minimized, it can be replaced by the equivalent problem of maximizing the same objective function with all its coefficients *cj*

replaced by $-cj$, $j = 1, 2, \dots, n$. If a constraint is given as an inequality, it can be replaced by an equivalent equation by adding a ***slack variable*** representing the difference between the two sides of the original inequality.

Thus, problem (10.2) in standard form is the following linear programming problem in four variables:

maximize $3x + 5y + 0u + 0v$

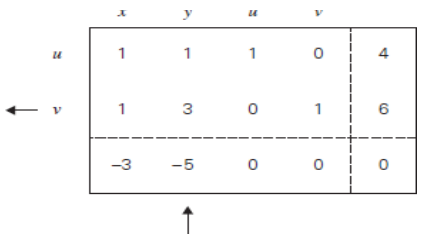
subject to $x + y + u = 4$

$x + 3y+ + v = 6$

$x, y, u, v \geq 0.$

We need to set two of the four variables in the constraint equations to zero to get a system of two linear equations in two unknowns and solve this system. For the general case of a problem with m equations in n unknowns $(n \geq m)$, $n - m$ variables need to be set to zero to get a system of m equations in m unknowns. If the system obtained has a unique solution— as any nondegenerate system of linear equations with the number of equations equal to the number of unknowns does—we have a ***basic solution***; its coordinates set to zero before solving the system are called ***nonbasic***, and its coordinates obtained by solving the system are called ***basic***.

If all the coordinates of a basic solution are nonnegative, the basic solution is called a ***basic feasible solution***. The simplex method progresses through a series of adjacent extreme points (basic feasible solutions) with increasing values of the objective function. Each such point can be represented by a ***simplex tableau***, a table storing the information about the basic feasible solution corresponding to the extreme point. For example, the


x
y
u
v
u
1
1
1
0
4
v
1
3
0
1
6
−3
−5
0
0
0

simplex tableau for *(0, 0, 4, 6)* of problem is presented below:

In general, a simplex tableau for a linear programming problem in standard form with n unknowns and m linear equality constraints $(n \geq m)$ has $m + 1$ rows and $n + 1$ columns. Each of the first m rows of the table contains the coefficients of a corresponding constraint equation, with the last column's entry containing the equation's right-hand side. The columns, except the last one, are labeled by the names of the variables. The last row of a simplex tableau is called the ***objective row***.

The objective row is used by the simplex method to check whether the current tableau represents an optimal solution: it does if all the entries in the objective row—except, possibly, the one in the last column—are nonnegative. If this is not the case, any of the negative entries indicates a nonbasic variable that can become basic in the next tableau.

A new basic variable is called the ***entering variable***, while its column is referred to as the ***pivot column***; we mark the pivot column by ↑ . *T*o get to an adjacent extreme point with a larger value of the objective function, we need to increase the entering variable by the largest amount possible to make one of the old basic variables zero while preserving the nonnegativity of all the others. We can translate this observation into the following rule for choosing a departing variable in a simplex tableau: for each *positive* entry in the pivot column, compute the ***θ-ratio*** by dividing the row's last entry by the entry in the pivot column. The row with the smallest θ-ratio determines the departing variable, i.e., the variable to become nonbasic.

Finally, the following steps need to be taken to transform a current tableau into the next one. (This transformation, called ***pivoting***). First, divide all the entries of the pivot row by the ***pivot***, its entry in the pivot column, to obtain row*new*←

rownew: 1/3 1 0 1/3 2.

Then, replace each of the other rows, including the objective row, by the difference **row – *c.* rownew**←, where *c* is the row's entry in the pivot column.

row 1 − 1 . rownew←: 2/3 0 1 − 1/3 2,

row 3 − *(−5)* . −−− rownew←: − 4/3 0 0 5/3 10. Thus,

the simplex method transforms tableau (10.5) into the following tableau:

	x	y	u	v	
← u	$\frac{2}{3}$	0	1	$-\frac{1}{3}$	2
y	$\frac{1}{3}$	1	0	$\frac{1}{3}$	2
	$-\frac{4}{3}$	0	0	$\frac{5}{3}$	10
	↑				

	x	y	u	v	
x	1	0	$\frac{3}{2}$	$-\frac{1}{2}$	3
y	0	1	$-\frac{1}{2}$	$\frac{1}{2}$	1
	0	0	2	1	14

This tableau represents the basic feasible solution *(*3, 1, 0, 0*)*. It is optimal because all the entries in the objective row of tableau are nonnegative. The maximal value of the objective function is equal to 14, the last entry in the objective row.

Summary of the simplex method:

Step 0 *Initialization* Present a given linear programming problem in standard form and set up an initial tableau with nonnegative entries in the rightmost column and *m* other columns composing the $m \times m$ identity matrix.

Step 1 *Optimality test* If all the entries in the objective row (except, possibly, the one in the rightmost column, which represents the value of the objective function) are nonnegative— stop: the tableau represents an optimal solution whose basic variables' values are in the rightmost column and the remaining, nonbasic variables' values are zeros.

Step 2 *Finding the entering variable* Select a negative entry from among the first *n* elements of the objective row.

Step 3 *Finding the departing variable* For each positive entry in the pivot column, calculate the

θ-ratio by dividing that row's entry in the rightmost column by its entry in the pivot

column. **Step 4** *Forming the next tableau* Divide all the entries in the pivot row by its entry in the pivot column. Subtract from each of the other rows, including the objective row, the new

pivot row multiplied by the entry in the pivot column of the row in question.

MAXIMUM-FLOW PROBLEM

The transportation network in question can be represented by a connected weighted digraph with

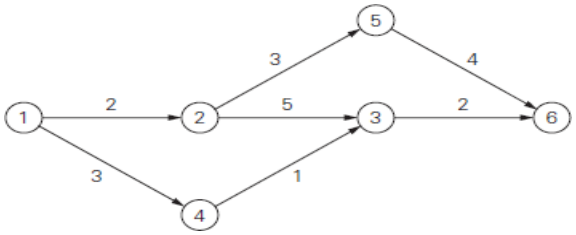
n vertices numbered from 1 to *n* and a set of edges *E,* with the following properties:

- It contains exactly one vertex with no entering edges; this vertex is called the ***source*** and assumed to be numbered 1.
- It contains exactly one vertex with no leaving edges; this vertex is called the ***sink*** and assumed to be numbered *n*.
- The weight *uij* of each directed edge *(i, j)* is a positive integer, called the edge ***capacity***. A digraph satisfying these properties is called a ***flow network*** or simply a ***network***.

The total amount of the material entering an intermediate vertex must be equal to the total amount of the material leaving the vertex. This condition is called the ***flow-conservation requirement***. If we denote the amount sent through edge *(i, j)* by *xij* , then for any intermediate vertex *i,* the flow-conservation requirement can be expressed by the following equality constraint:

$$\sum_{j:(j,i)\in E} x_{ji} = \sum_{j:(i,j)\in E} x_{ij} \quad \text{for } i = 2, 3, \ldots, n-1,$$

where the sums in the left- and right-hand sides express the total inflow and outflow entering and leaving vertex *i,* respectively. Since no amount of the material can change by going through intermediate vertices of the network, it stands to reason that the total amount of the material leaving the source must end up at the sink.



Thus, we have the following equality:

$$\sum_{j:(1,j)\in E} x_{1j} = \sum_{j:(j,n)\in E} x_{jn}.$$

This quantity, the total outflow from the source—or, equivalently, the total inflow into the sink—is called the ***value*** of the flow.We denote it by *v*. It is this quantity that we will want

to maximize over all possible flows in a network. Thus, a (feasible) ***flow*** is an assignment of real numbers *xij* to edges *(i, j)* of a given network that satisfy flow-conservation constraints (10.8) and the ***capacity constraints***

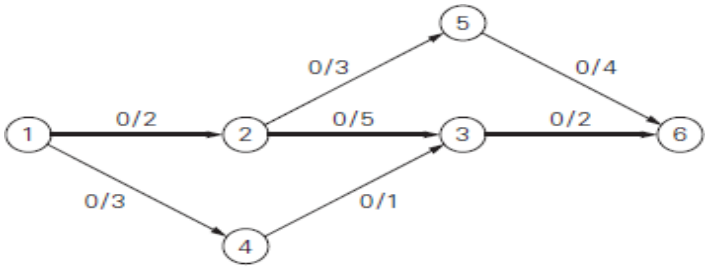
$$0 \le x_{ij} \le u_{ij} \quad \text{for every edge } (i, j) \in E.$$

The ***maximum-flow problem*** can be stated formally as the following optimization problem:

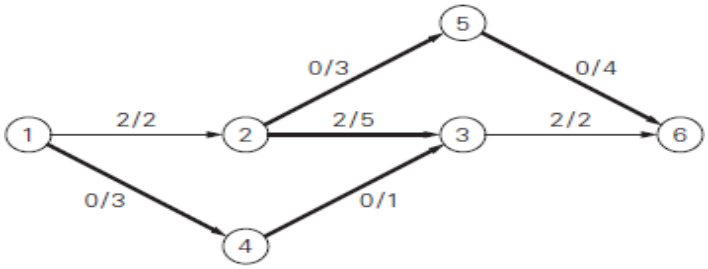
$$\begin{aligned} \text{maximize} \quad & v = \sum_{j:(1,j)\in E} x_{1j} \\ \text{subject to} \quad & \sum_{j:(j,i)\in E} x_{ji} - \sum_{j:(i,j)\in E} x_{ij} = 0 \quad \text{for } i = 2, 3, \ldots, n-1 \\ & 0 \le x_{ij} \le u_{ij} \quad \text{for every edge } (i, j) \in E. \end{aligned}$$

- An actual implementation of the augmenting path idea is, however, not quite straightforward. To see this, let us consider the network in Figure. We start with the zero flow shown in Figure.
- Among several possibilities, let us assume that we identify the augmenting path 1→2→3→6 first. We can increase the flow along this path by a maximum of 2 units, which is the smallest unused capacity of its edges.
- The new flow is shown in Figure 10.5b. This is as far as our simpleminded idea about flow-augmenting paths will be able to take us. Unfortunately, the flow shown in Figure is not optimal: its value can still be increased along the path 1→4→3←2→5→6 by increasing the flow by 1 on edges *(1, 4)*, *(4, 3)*, *(2, 5)*, and *(5,6)* and *decreasing* it by 1 on edge *(2, 3)*.
- The flow obtained as the result of this augmentation is shown in Figure. It is indeed maximal.
- Thus, to find a flow-augmenting path for a flow *x,* we need to consider paths from source to sink in the underlying *undirected* graph in which any two consecutive vertices *i, j* are either i. connected by a directed edge from *i* to *j* with some positive unused capacity *rij = uij − xij* (so that we can increase the flow through that edge by up to *rij* units), or ii. connected by a directed edge

- Edges of the first kind are called ***forward edges*** because their tail is listed before their head in the vertex list $1 \rightarrow \ldots i \rightarrow j \ldots \rightarrow n$ defining the path; edges of the second kind are called ***backward edges*** because their tail is listed after their head in the path list $1 \rightarrow \ldots i \leftarrow j \ldots \rightarrow n$.

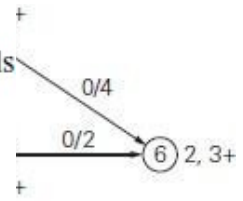


(a)

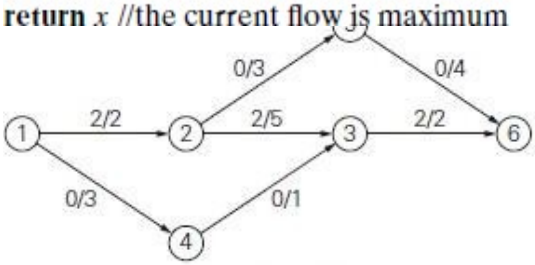


(b)

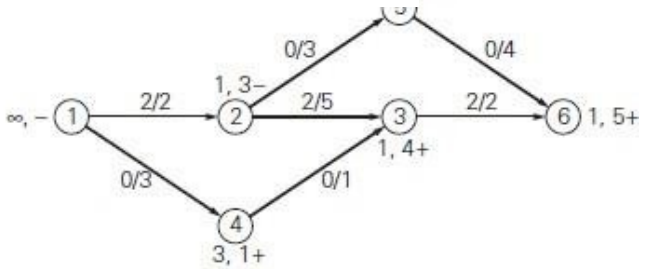
for every edge from j to i **do** //backward edges
  if j is unlabeled
    if $x_{ji} > 0$
      $l_j \leftarrow \min\{l_i, x_{ji}\}$; label j with l_j, i^-
      $Enqueue(Q, j)$
if the sink has been labeled
  //augment along the augmenting path found
  $j \leftarrow n$ //start at the sink and move backwards using second labels
  while $j \neq 1$ //the source hasn't been reached
    if the second label of vertex j is i^+
      $x_{ij} \leftarrow x_{ij} + l_n$
    else //the second label of vertex j is i^-
      $x_{ji} \leftarrow x_{ji} - l_n$
    $j \leftarrow i$; $i \leftarrow$ the vertex indicated by i's second label
  erase all vertex labels except the ones of the source
  reinitialize Q with the source
return x //the current flow is maximum



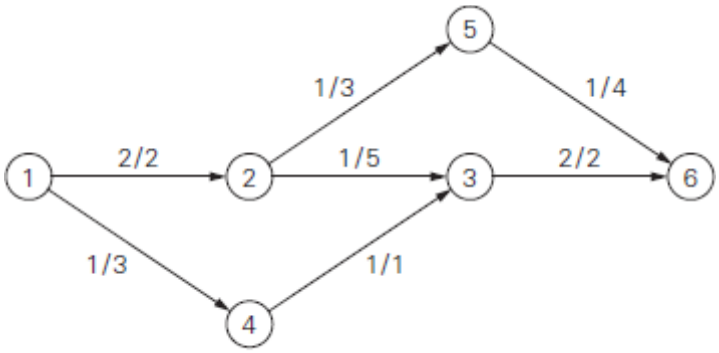
's first label)
$3 \rightarrow 6$.



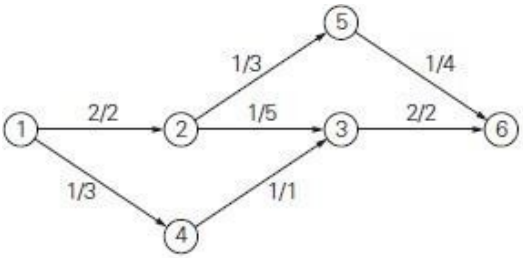
Queue: 1 4 3 2 5 6
↑ ↑ ↑ ↑ ↑



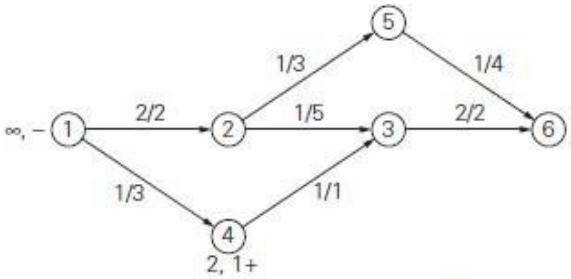
Augment the flow by 1 (the sink's first label)
along the path $1 \rightarrow 4 \rightarrow 3 \leftarrow 2 \rightarrow 5 \rightarrow 6$.



(c)



Queue: 1 4
↑ ↑



No augmenting path (the sink is unlabeled);
the current flow is maximal.

MAXIMUM MATCHING IN BIPARTITE GRAPHS

A ***matching*** in a graph is a subset of its edges with the property that no two edges share a vertex. A ***maximum matching***—more precisely, a ***maximum cardinality matching***—is a matching with the largest number of edges. In a ***bipartite graph***, all the vertices can be partitioned into two disjoint sets *V* and *U*, not necessarily of the same size, so that every edge connects a vertex in one of these sets to a vertex in the other set.

Let us apply the iterative-improvement technique to the maximum cardinality-matching problem. Let *M* be a matching in a bipartite graph *G* ={*V, U, E}*. How can we improve it, i.e., find a new matching with more edges? Obviously, if every vertex in either *V* or *U* is ***matched*** (has a ***mate***), i.e., serves as an endpoint of an edge in *M*, this cannot be done and *M* is a maximum matching.

Therefore, to have a chance at improving the current matching, both *V* and *U* must contain ***unmatched*** (also called ***free***) ***vertices***, i.e., vertices that are not incident to any edge in *M*. For example, for the matching *Ma* = {*(*4, 8*), (*5, 9*)*} in the graph in Figure vertices 1, 2, 3, 6, 7, and 10 are free, and vertices 4, 5, 8, and 9 are matched.

Another obvious observation is that we can immediately increase a current matching by adding an edge between two free vertices. For example, adding *(*1, 6*)* to the matching *Ma*

= {*(*4, 8*), (*5, 9*)*} in the graph in Figure 10.9a yields a larger matching *Mb* = {*(*1, 6*), (*4, 8*),*

*(*5, 9*)*}

Let us now try to find a matching larger than *Mb* by matching vertex 2. The only way to do this would be to include the edge *(*2, 6*)* in a new matching. This inclusion requires removal of *(*1, 6*)*, which can be compensated by inclusion of *(*1, 7*)* in the new matching. This new matching *Mc* = {*(*1, 7*), (*2, 6*), (*4, 8*), (*5, 9*)*} is shown in Figure

In general, we increase the size of a current matching *M* by constructing a simple path from a free vertex in *V* to a free vertex in*U* whose edges are alternately in *E* −*M* and in

M. That is, the first edge of the path does not belong to *M,* the second one does, and so on, until the last edge that does not belong to *M.* Such a path is called ***augmenting*** with respect to the matching

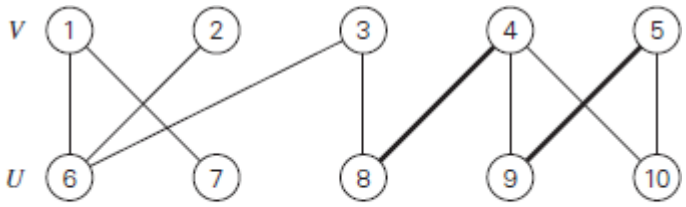
For example, the path 2, 6, 1, 7 is an augmenting path with respect to the matching *Mb* in Figure. Since the length of an augmenting path is always odd, adding to the

matching *M* the path's edges in the odd-numbered positions and deleting from it the path's edges in the even- numbered positions yields a matching with one more edge than in *M*.

Such a matching adjustment is called ***augmentation***. Thus, in Figure the matching *Mb* was obtained by augmentation of the matching *Ma* along the augmenting path 1, 6, the matching *Mc* was obtained by augmentation of the matching *Mb* along the augmenting path 2, 6, 1, 7. Moving further, 3, 8, 4, 9, 5, 10 is an augmenting path for the matching *Mc* After

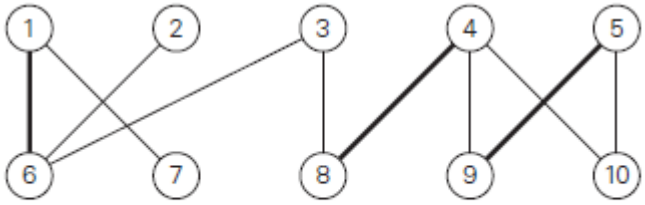
adding to *Mc* the edges *(*3, 8*)*, *(*4, 9*)*, and *(*5, 10*)* and deleting *(*4, 8*)* and *(*5, 9*)*, we obtain

the matching *Md* = {*(*1, 7*)*, *(*2, 6*)*, *(*3, 8*)*, *(*4, 9*)*, *(*5, 10*)*} shown in Figure 10.9d. The matching *Md* is not only a maximum matching but also ***perfect***, i.e., a matching that matches all the vertices of the graph.



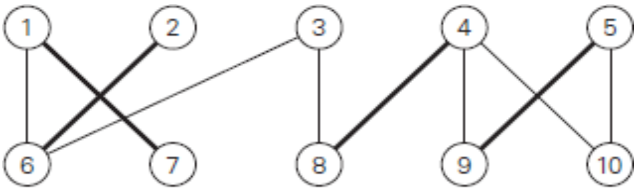
(a)

Augmenting path: 1, 6



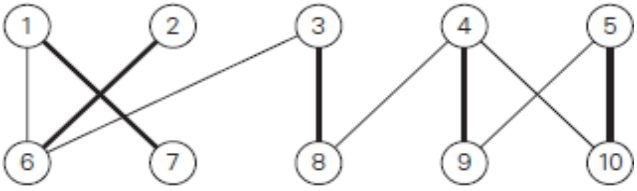
(b)

Augmenting path: 2, 6, 1, 7



(c)

Augmenting path: 3, 8, 4, 9, 5, 10

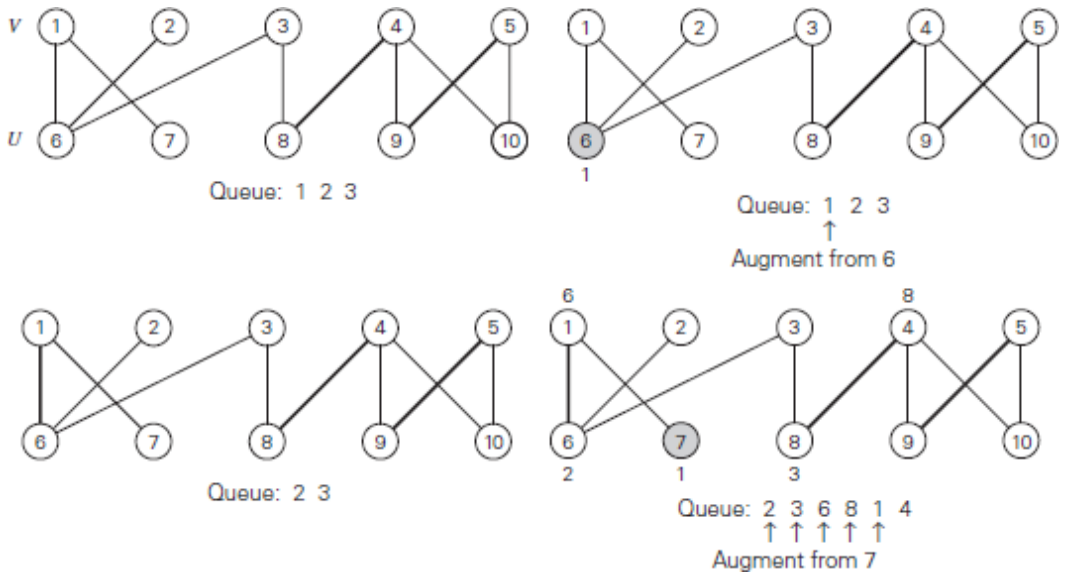


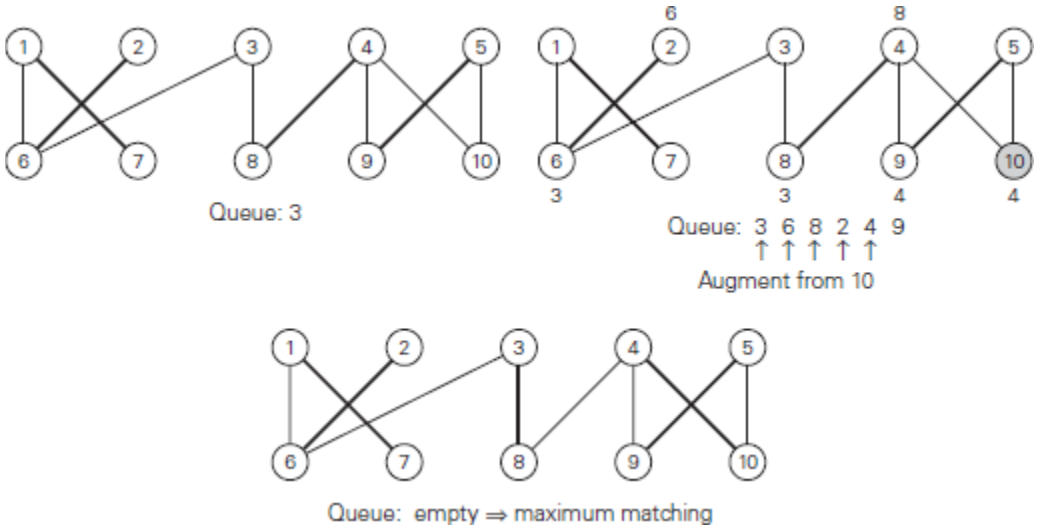
(d)

Maximum matching

ALGORITHM *MaximumBipartiteMatching(G)*

```
//Finds a maximum matching in a bipartite graph by a BFS-like traversal
//Input: A bipartite graph G = ⟨V, U, E⟩
//Output: A maximum-cardinality matching M in the input graph
initialize set M of edges with some valid matching (e.g., the empty set)
initialize queue Q with all the free vertices in V (in any order)
while not Empty(Q) do
    w ← Front(Q);  Dequeue(Q)
    if w ∈ V
        for every vertex u adjacent to w do
            if u is free
                //augment
                M ← M ∪ (w, u)
                v ← w
                while v is labeled do
                    u ← vertex indicated by v's label;  M ← M − (v, u)
                    v ← vertex indicated by u's label;  M ← M ∪ (v, u)
                remove all vertex labels
                reinitialize Q with all free vertices in V
                break  //exit the for loop
            else  //u is matched
                if (w, u) ∉ M and u is unlabeled
                    label u with w
                    Enqueue(Q, u)
    else  //w ∈ U (and matched)
        label the mate v of w with w
        Enqueue(Q, v)
return M  //current matching is maximum
```





THE STABLE MARRIAGE PROBLEM

In this section, we consider an interesting version of bipartite matching called the stable marriage problem. Consider a set $Y = \{m1, m2, \ldots, mn\}$ of n men and a set $X = \{w1, w2, \ldots, wn\}$ of n women. Each man has a preference list ordering the women as potential marriage partners with no ties allowed. Similarly, each woman has a preference list of the men, also with no ties. Examples of these two sets of lists are given in Figures.

The same information can also be presented by an $n \times n$ ranking matrix (see Figure The rows and columns of the matrix represent the men and women of the two sets, respectively. A cell in row m and column w contains two rankings: the first is the position (ranking) of w in the m's preference list; the second is the position (ranking) of m in the w's preference list.

For example, the pair 3, 1 in Jim's row and Ann's column in the matrix in Figure indicates that Ann is Jim's third choice while Jim is Ann's first. Which of these two ways to represent such information is better depends on the task at hand. For example, it is easier to specify a match of the sets' elements by using the ranking matrix, whereas the preference lists might be a more efficient data structure for implementing a matching algorithm.

A ***marriage matching*** M is a set of n (m, w) pairs whose members are selected

from disjoint *n*-element sets *Y* and *X* in a one-one fashion, i.e., each man *m* from *Y* is paired with exactly one woman *w* from *X* and vice versa. (If we represent *Y* and *X* as vertices of a complete bipartite graph with edges connecting possible marriage partners, then a marriage

matching is a perfect matching in such a graph.)

men's preferences

	1st	2nd	3rd
Bob:	Lea	Ann	Sue
Jim:	Lea	Sue	Ann
Tom:	Sue	Lea	Ann

(a)

women's preferences

	1st	2nd	3rd
Ann:	Jim	Tom	Bob
Lea:	Tom	Bob	Jim
Sue:	Jim	Tom	Bob

(b)

ranking matrix

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

(c)

A pair *(m, w),* where *m* ∈ *Y, w* ∈ *X,* is said to be a ***blocking pair*** for a marriage matching *M* if man *m* and woman *w* are not matched in *M* but they prefer each other to their mates in *M.* For example, (Bob, Lea) is a blocking pair for the marriage matching *M* = {(Bob, Ann), (Jim, Lea), (Tom, Sue}}because they are not matched in *M* while Bob prefers Lea to Ann and Lea prefers Bob to Jim. A marriage matching *M* is called ***stable*** if there is no blocking pair for it; otherwise, *M* is called ***unstable***. According to this definition, the marriage matching in Figure is unstable because Bob and Lea can drop their designated mates to join in a union they both prefer. The ***stable marriage problem*** is to find a stable marriage matching for men's and women's given preferences. Surprisingly, this problem always has a solution.

Stable marriage algorithm

Input: A set of *n* men and a set of *n* women along with rankings of the women by each man and rankings of the men by each woman with no ties allowed in the rankings

Output: A stable marriage matching

Step 0 Start with all the men and women being free.

Step 1 While there are free men, arbitrarily select one of them and do the following:

Proposal The selected free man *m* proposes to *w,* the next woman on his preference list (who is the highest-ranked woman who has not rejected him before).

Response If *w* is free, she accepts the proposal to be matched with *m*. If she is not free, she compares *m* with her current mate. If she prefers*m* to him, she accepts *m*'s proposal, making her former mate free; otherwise, she simply rejects *m*'s proposal, leaving *m* free.

Step 2 Return the set of *n* matched pairs.

Before we analyze this algorithm, it is useful to trace it on some input. Such an example is presented in Figure.

Let us discuss properties of the stable marriage algorithm.

		Ann	Lea	Sue	
Free men: Bob, Jim, Tom	Bob	2, 3	[1,2]	3, 3	Bob proposed to Lea Lea accepted
	Jim	3, 1	1, 3	2, 1	
	Tom	3, 2	2, 1	1, 2	
Free men: Jim, Tom	Bob	2, 3	[1,2]	3, 3	Jim proposed to Lea Lea rejected
	Jim	3, 1	1, 3	2, 1	
	Tom	3, 2	2, 1	1, 2	
Free men: Jim, Tom	Bob	2, 3	[1,2]	3, 3	Jim proposed to Sue Sue accepted
	Jim	3, 1	1, 3	[2,1]	
	Tom	3, 2	2, 1	1, 2	
Free men: Tom	Bob	2, 3	[1,2]	3, 3	Tom proposed to Sue Sue rejected
	Jim	3, 1	1, 3	[2,1]	
	Tom	3, 2	2, 1	1, 2	
Free men: Tom	Bob	2, 3	1, 2	3, 3	Tom proposed to Lea Lea replaced Bob with Tom
	Jim	3, 1	1, 3	[2,1]	
	Tom	3, 2	[2,1]	1, 2	
Free men: Bob	Bob	[2,3]	1, 2	3, 3	Bob proposed to Ann Ann accepted
	Jim	3, 1	1, 3	[2,1]	
	Tom	3, 2	[2,1]	1, 2	

Iterative Improvement Technique

Iterative improvement is a problem-solving technique used in Design and Analysis of Algorithms (DAA) where a solution is progressively refined through repeated modifications to an initial feasible solution until an optimal or near-optimal solution is achieved. This method is particularly useful for optimization problems, where the goal is to find the best solution (minimum or maximum) from a set of feasible solutions.

Concept:

1. Initialization:

    Start with an initial feasible solution to the problem. This solution doesn't need to be optimal, but it should be a valid solution within the problem's constraints.

2. Iteration:

    Improvement Step:

Evaluate the current solution and identify potential improvements. These improvements usually involve small, local changes to the solution.

    Selection:

Choose the best improvement among the possible ones, based on the problem's objective function (e.g., minimizing cost or maximizing profit).

    Update:

Apply the chosen improvement to the current solution, creating a new, potentially better solution.

3. Termination:

    Repeat the iteration step until no further improvements can be found. This point is often referred to as a "local optimum" or "near-optimal solution".

    The algorithm may also terminate after a set number of iterations or when a certain solution quality threshold is met.

Key aspects:

    Feasible Region:

Iterative improvement algorithms explore the feasible region of the problem, which is the set of all possible solutions that satisfy the constraints.

    Local vs. Global Optimum:

The algorithm might find a solution that is the best within its immediate neighborhood (local optimum) but not necessarily the absolute best solution for the entire problem (global optimum).

Examples:

- Simplex Method: For linear programming problems, the Simplex method iteratively moves from one basic feasible solution to another, improving the objective function value until an optimal solution is reached.
- Maximum Flow (Ford-Fulkerson): The Ford-Fulkerson algorithm iteratively finds augmenting paths in a flow network and increases the flow along these paths until the maximum flow is achieved.
- Local Search Heuristics: Algorithms like hill climbing and simulated annealing use iterative improvement to find good solutions to complex problems by exploring the solution space and making small adjustments.

Advantages:

Simplicity:

Iterative improvement algorithms are often relatively easy to understand and implement.

Efficiency:

They can be efficient for certain types of problems, especially when good initial solutions are available.

Disadvantages:

- Local Optima: They might get stuck in local optima, failing to find the global optimum.
- Sensitivity to Initial Solution: The quality of the final solution can be influenced by the initial solution.