# Fundamentals of Algorithms*

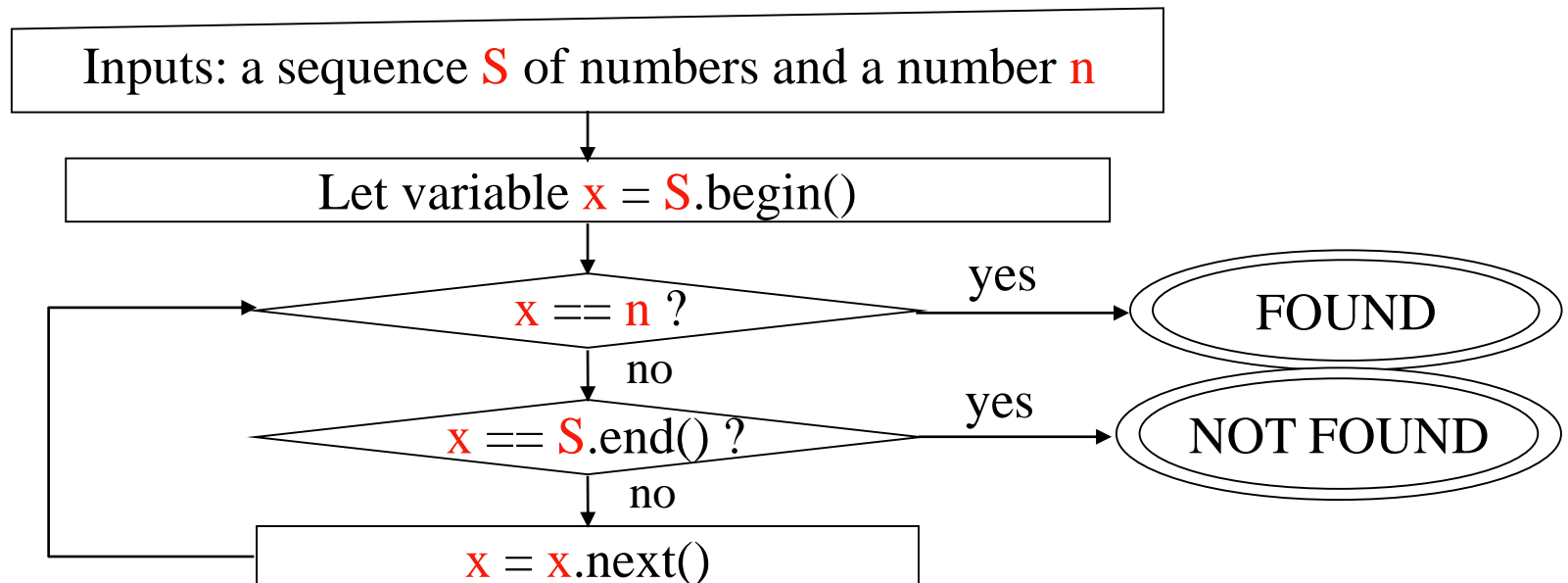* Slides are based on Chapter 4 of the following book:
Electronic Design Automation: Synthesis, Verification, and Test,  edited by L.-T. Wang, Y.-W. Chang, K.-T. Cheng, Morgan Kaufmann, 2009.

# Algorithm

- What is an "algorithm"?

  – A sequence of well-defined instructions for completing a task or solving a problem

  – Can be described in a natural language, pseudo code, a flow chart, or even a programming language

# Example of an Algorithm

- Problem
  - Whether a specific number is contained in a given sequence of numbers

- Algorithm: Linear Search

Inputs: a sequence S of numbers and a number n

Let variable x = S.begin()

x == n ?   → yes → FOUND

no

x == S.end() ?   → yes → NOT FOUND

no

x = x.next()

# Evaluation of an Algorithm

- Is the algorithm in the previous slide efficient?
  - Can we make a better algorithm?
- Can the algorithm complete the task within an acceptable amount of time for a specific set of data derived from a practical application?
  - How to quantify the "efficiency" of an algorithm?

# Some Well-Developed Algorithms

- Graph algorithms
  - Converting problems into graphs, and using existing graph algorithms to solve them
  - Breadth-First Search and Depth-First Search, Topological Sort, Shortest and Longest Path Algorithms, etc.
- Heuristic algorithms
  - Fact: Many (or most) EDA problems are intrinsically very difficult because finding an optimal solution within a reasonable runtime is not always possible
  - ➔ Find an acceptable solution first. If time or computer resources permit, further improve the result incrementally
- Mathematical programming
  - Transforming problems into certain mathematical models, such as linear inequalities or non-linear equations
  - Well developed algorithms or even tools are available to solve these mathematical formulae very efficiently

# Computational Complexity

- Computational Complexity
  - A measurement of efficiency of algorithms
  - Measured with *memory* and *time* used
    - Space complexity: used memory
    - Time complexity: used time, more important than space complexity
  - A mathematical function of the *input size*
    - Sorting *n* words: input size *n*
    - Graph: #vertices (*|V|*) and #edges (*|E|*)

# Average- and Worst-Case Complexity

- Time complexity
  - Amount of required elementary computational steps
  - Differs from input to input because of the existence of conditional constructs
    - If-else statements
  - Average-case complexity is needed to leverage the differences among inputs
    - Hard to compute, however
    - Use *worst-case complexity* instead in practice

# Asymptotic Notations

- Asymptotic functions
  - Expressions of complexity, or runtime
  - Only care about the rate of growth, or the order of growth
  - When expressed in functions of the input size $n$
    - Lower-order terms ignored, e.g., $n^2 + n \rightarrow n^2$
    - Coefficients ignored, e.g., $3n \rightarrow n$
  - Most used notations
    - $O$, $\Theta$ and $\Omega$

# O-notation

- *O*: upper bounds of complexity functions
- Definition:
  - $O(g(n)) = \{ f(n) : c, n_0 > 0$ exist such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0 \}$
  - Conventionally written as $f(n) = O(g(n))$, but it's NOT equivalence
- Examples
  - $n^3 + 1000n^2 + n = O(n^3)$
  - $n^2 = O(2^n)$

# Common O-notations

- Polynomial-time complexity: $O(n^k)$, where $n$ is the input size and $k$ is a constant
  - $O(1)$: constant time
  - $O(\lg n)$: logarithmic time
  - $O(n)$: linear time
  - $O(n^2)$: quadratic time
  - $O(n^3)$: cubic time
- Non-polynomial time complexity:
  - $O(2^n)$, $O(3^n)$: exponential time
  - $O(n!)$: factorial time

# Ω-notation

- Ω-notation
  - the inverse of *O*-notation
    - $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$
  - lower bounds of complexity functions
- Definition:
  - $\Omega(g(n)) = \{ f(n) : c, n_0 > 0$ exist such that $0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0 \}$
- Much less common than *O*-notation

# $\Theta$-notation

- $\Theta$-notation

  - tight bounds of complexity functions

    - $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

- Definition:

  - $\Theta(g(n)) = \{ f(n) : c_1, c_2, n_0 > 0 \text{ exist such that } 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ for all } n \geq n_0 \}$

- Examples

  - $0.1n^2 + 1000n = \Theta(n^2)$

  - $10000n^2 + 2^n = \Theta(2^n)$

# Decision Problems

- Decision problems
  - Can only be answered with "yes" or "no"
  - Names in capital letters

- HAMILTONIAN CYCLE
  - Given a graph, whether a loop that goes through all nodes exists
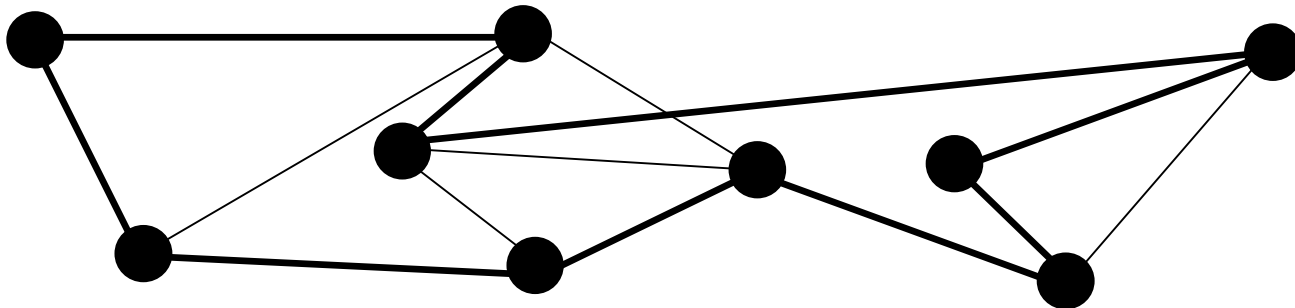
# Optimization Problems

- An optimization problem
  - Looks for an optimum of the objective function
  - Can be decomposed into a series of decision problems by binary search
- Traveling Salesman Problem (TSP)
  - Finding a shortest route for a salesman who needs to visit a set of cities in a round tour

(a)  (b)  (c)
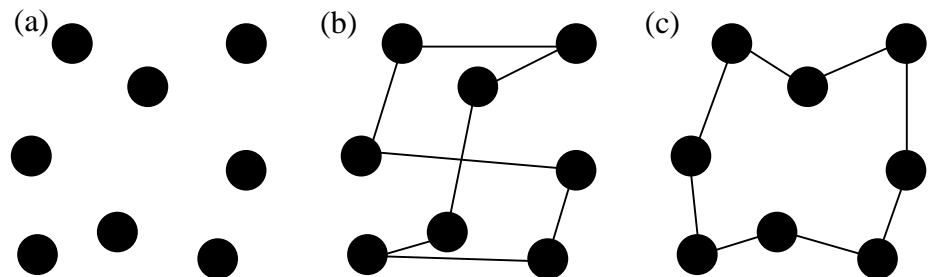
# Complexity Classes

- Complexity classes
  - A set of problems with the same degree of complexity, where the complexity of a problem is the complexity of its most efficient possible algorithm
- 4 most frequently seen classes
  - P (Polynomial)
  - NP (Nondeterministic Polynomial)
  - NP-complete
  - NP-hard

# Complexity Class P

- Complexity Class P
  - P stands for polynomial
  - Contains the problems that can be solved in polynomial time in terms of the input size on a deterministic computer
  - For a problem in P with input size $n$, an algorithm of complexity $O(n^k)$ exists, where $k$ is a constant
  - Problems in P are considered *tractable*

# Complexity Class NP

- Problems in class NP (Nondeterministic Polynomial)
  - Decision problems
  - Can be solved in polynomial time on a *nondeterministic* computer
  - The solutions can be verified for correctness in polynomial time on a *deterministic* computer
- P is contained in NP
- But the question of if P = NP remains open

# Polynomial Transformation

- Polynomial transformation from problems $P_a$ to $P_b$
  - Expresses an instance of $P_a$ as an instance of $P_b$
  - The transformation is in polynomial time complexity
- $P_a$ is polynomially reducible to $P_b$
  - A polynomial transformation (or reduction) from $P_a$ to $P_b$ exists
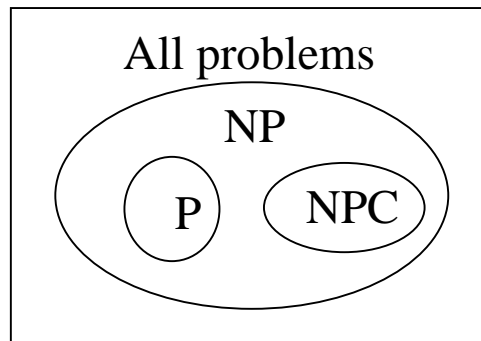
# Complexity Class NP-complete (NPC)

- Informal definition
  - The most difficult problems in class NP
- Formal definition
  - The problem $P_b$ is in NPC if
    - $P_b$ is in NP
    - For any problem $P_a$ in NP, a *polynomial transformation* from $P_a$ to $P_b$ always exists
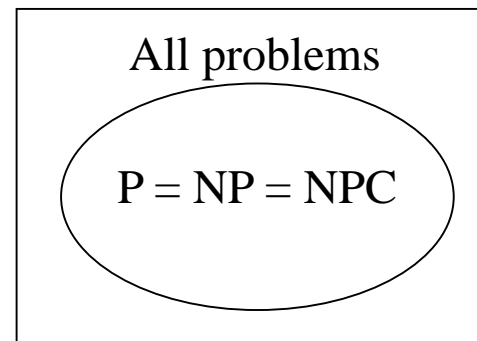- Problems in NPC are *polynomially reducible* to one another

# Complexity Class NPC

- Checking if a problem $P_a$ is in NPC
  - $P_a$ is in NP
    - Solution checking can be done in polynomial time
  - A known NPC problem is polynomially reducible to $P_a$
- P = NP iff P = NPC
  - Still an open question

(a) P ≠ NP

All problems

NP

P      NPC

(b) P = NP

All problems

P = NP = NPC

# Complexity Class NP-hard

- NP-hard problems
  - At least as hard as NPC problems (most of the time harder)
  - Solution checking cannot be done in polynomial time
- In practice, most optimization versions of NPC problems are NP-hard
  - TSP as the optimization version of TRAVELING SALESMAN: if a round tour of length under a constant $k$ exists
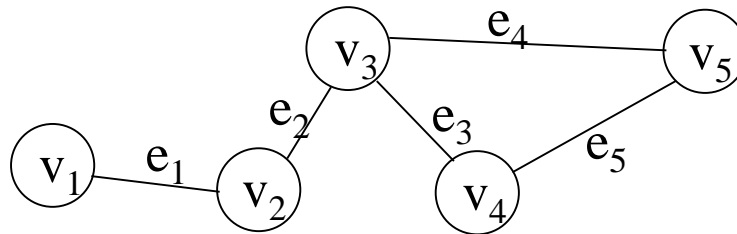
# Graph

- Graph
  - Models pairwise relationships among items of certain form
  - We can model circuits with directed graphs
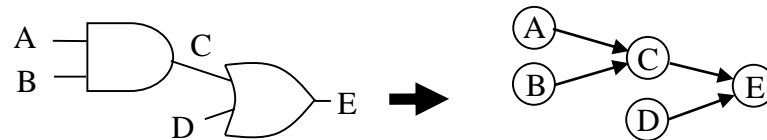
  

  - Defined by 2 sets: a vertex (node) set $V$ and an edge set $E$
    - Edges can be either directed or undirected

# Graph Terminology



- An undirected graph $G = (V, E)$
  - $V = \{v_1, v_2, v_3, v_4, v_5\}$, $|V| = 5$
  - $E = \{e_1, e_2, e_3, e_4, e_5\}$, $|E| = 5$
  - An edge has 2 endpoints, e.g., $e_1 = (v_1, v_2)$
    - $v_1$ and $v_2$ are adjacent
    - $e_1$ is incident with $v_1$ and $v_2$
  - Degree of a vertex: number of incident edges
    - degree($v_3$) = 3, degree($v_5$) = 2

# Graph Terminology (Cont'd)
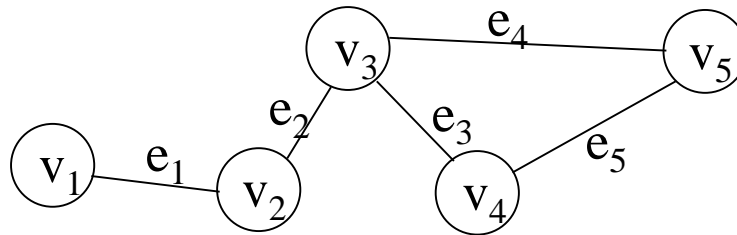


- Path: $\langle v_5, e_4, v_3, e_3, v_4 \rangle$, its length (number of edges in the path) is 2

- Cycle: $\langle v_5, e_4, v_3, e_3, v_4, e_5, v_5 \rangle$ starts and ends at the same vertex

- Simple path: no cycles in the path

# Graph Terminology (Cont'd)

- Loop
  - An edge starting and ending at the same vertex
- Parallel edges
  - Plural edges incident with the same two vertices
- Simple graph: no loops or parallel edges
- $|V|$ and $|E|$ can be simply $V$ and $E$ inside asymptotic notations, e.g. $O(V + E)$
- Weighted graphs
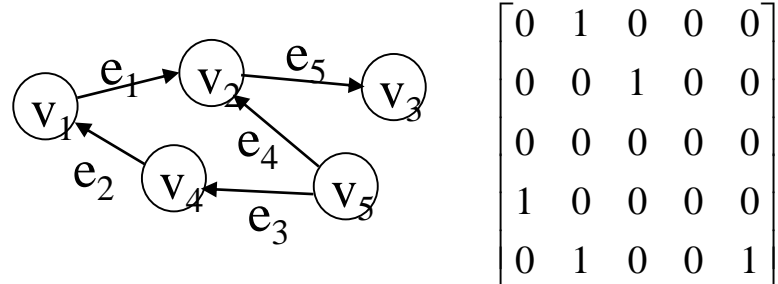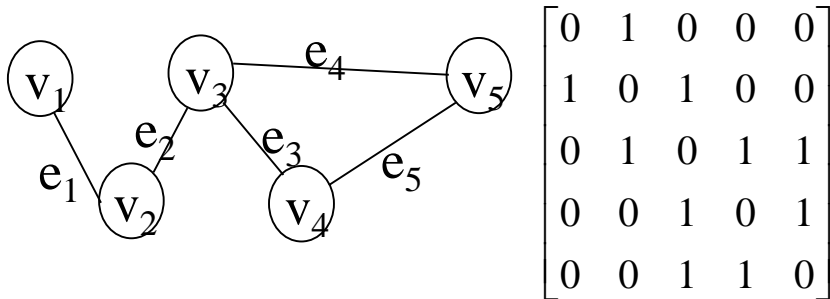  - Various values (weights) are assigned to edges

# Terminology for Directed Graphs

- Edge: $e_1 = (v_1, v_2) \neq (v_2, v_1)$
  - $e_1$ is incident from $v_1$ *to* $v_2$
  - $v_2$ is the head of $e_1$
  - $v_1$ is the tail of $e_1$

- Path: $\langle v_4, e_4, v_2, e_2, v_3 \rangle$
  - $v_4$ appears before $v_3$ in the path
  - $v_4$ is $v_3$'s predecessor; $v_3$ is $v_4$'s successor

- DAG: directed acyclic graphs
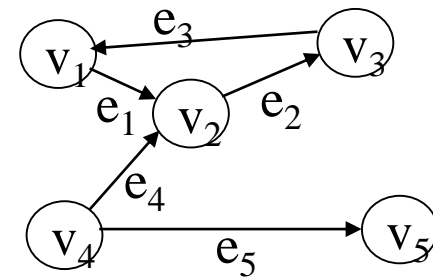
# Representation: Adjacency Matrix

- A $|V|$ x $|V|$ matrix
  - $A_{mn} = 1$ if $(v_m, v_n) \in E$
  - $A_{mn} = 0$ if $(v_m, v_n) \notin E$
- Symmetric for undirected graphs
- $\Theta(V^2)$ space, efficient for dense graphs



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

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

# Representation: Adjacency List

- An array of |$V$| linked lists each of which stores all the adjacent vertices to a respective vertex in $V$
- $\Theta(V + E)$ space, efficient for sparse graphs
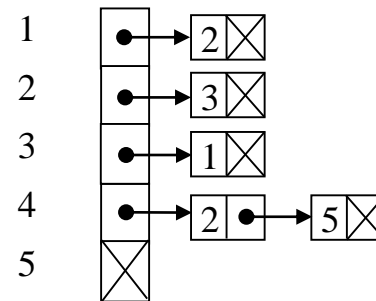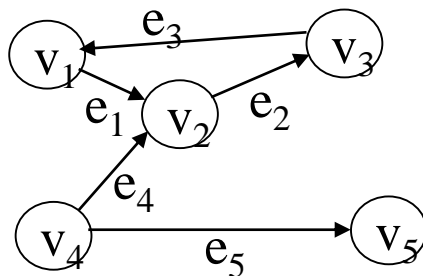
# Breadth-First Search (BFS)

```
BFS(Graph G, Vertex s)
1   FIFO_Queue Q = {s};
2   for (each v ∈ V){
3       v.visited = false;
4       v.distance = ∞;
5       v.predecessor = NIL;
6   }
7   s.visited = true;
8   s.distance = 0;
9   while (Q ≠ ∅){
10      Vertex u = Dequeue(Q);
11      for (each (u, w) ∈ E){
12         if (!(w.visited)){
13            w.visited = true;
14            w.distance = u.distance + 1;
15            w.predecessor = u;
16            Enqueue(Q, w);
17         }
18      }
19  }
```

- BFS: visit neighbors of the source first, then neighbors of neighbors, and so on

- *s:* source of the graph

- distance: length of the shortest path from source to the vertex

# Breadth-First Search (BFS) (Cont'd)
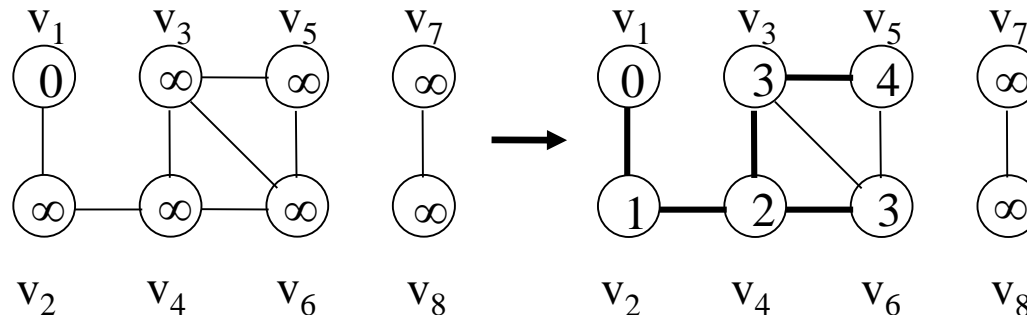


- BFS
  - Builds a breadth-first tree (thick edges)
  - Finds connected components (nodes with finite distances)
  - Finds shortest paths to the source in unweighted graphs

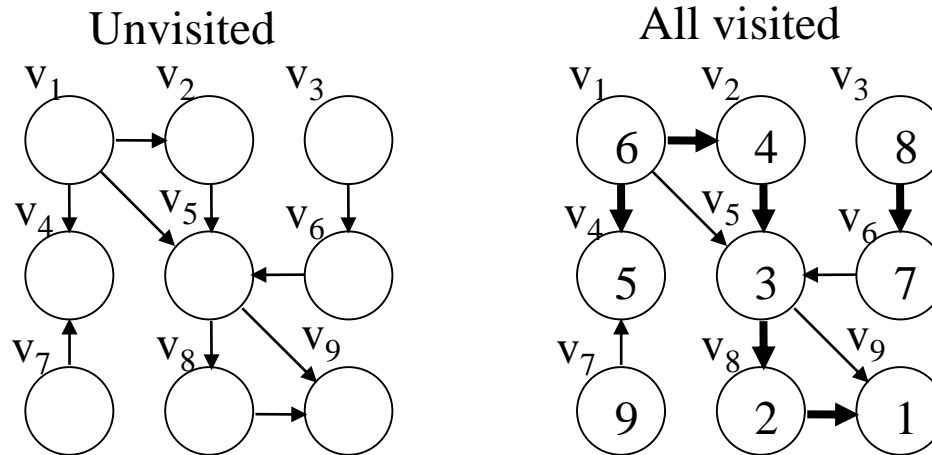# Depth-First Search (DFS)

```
DFS(Graph G)
1   for (each vertex v ∈ V){
2       v.visited = false;
3       v.predecessor = NIL;
4   }
5   time = 0;
6   for (each vertex v ∈ V)
7       if (!(v.visited)) DFSVisit(v);


DFSVisit(Vertex v)
1   v.visited = true;
2   for (each (v, u) ∈ E){
3       if (!(u.visited)){
4           u.predecessor = v;
5           DFSVisit(u);
6       }
7   }
8   time = time + 1;
9   v.PostOrderTime = time;
```

- DFS: traverse as deeply as possible before backtracking
- PostOrderTime: the visiting sequence of each node in post order
- Time complexity:

  $O(V + E)$

# Depth-First Search (DFS) (Cont'd)

Unvisited

$v_1$  $v_2$  $v_3$

$v_4$  $v_5$  $v_6$

$v_7$  $v_8$  $v_9$

All visited

$v_1$ 6  $v_2$ 4  $v_3$ 8

$v_4$ 5  $v_5$ 3  $v_6$ 7

$v_7$ 9  $v_8$ 2  $v_9$ 1

- Values in the nodes: PostOrderTime
- Guaranteed to visit all vertices
  - Forms a depth-first forest (instead of a tree)
- Important applications
  - E.g., topological sort

Unit 2

32

# Topological Sort

- Applied on DAGs

- Gives a linear ordering of vertices, where no vertex appears before its predecessors

- $O(V + E)$ complexity (because of DFS)

TopologicalSortByDFS(Graph G)
1    call DFS(G), and insert v onto the front of a linked list L as PostOrderTime of each vertex v is computed;
2    return L;

Topological sort of the graph on the previous page
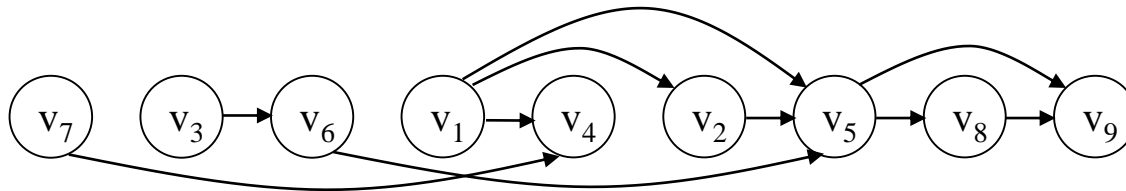
# Strongly Connected Components

- Strongly connected graphs
  - Directed
  - All vertices can reach each other through directed edges

- Strongly connected components (SCC)
  - Partitioning of a directed graph
  - Each partition is
    - Strongly connected
    - Maximal, no vertex can be added
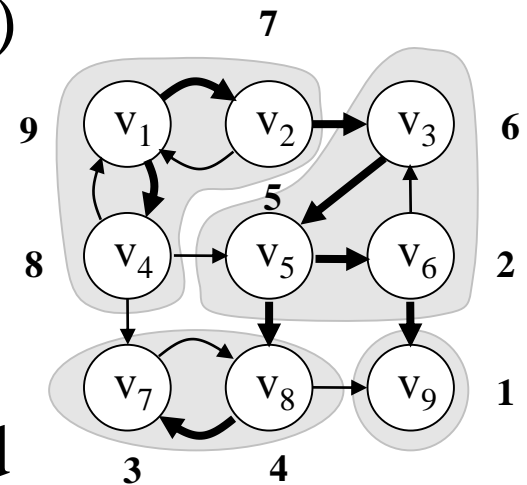
# Strongly Connected Components

SCC(Graph G)
1   call DFS(G) for PostOrderTime;
2   $G^T$ = transpose(G);
3   call DFS($G^T$), replacing line 6 of DFS with a procedure examining vertices in
       order of decreasing PostOrderTime;
4   return different trees in depth-first forest built in DFS($G^T$) as separate SCCs;

- Algorithm:

   – A DFS, a transpose, then another DFS

   – $O(V + E)$ time complexity (because of DFS)

- Resultant SCCs form a DAG
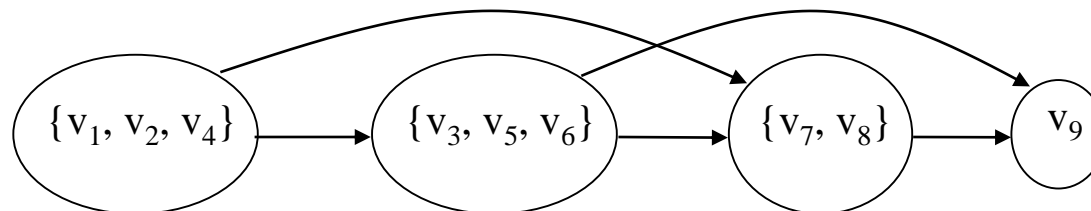
# Shortest Path Algorithms

- Applied on directed, weighted graphs
- Different algorithms for different graphs
  - DAG shortest path algorithm: on DAGs
  - Dijkstra's Algorithm: no negative weights
  - The Bellman-Ford Algorithm: most general
- All 3 algorithms share the same kernel
  - initialization and relaxation

# Initialization and Relaxation

Initialize(graph G, Vertex s)
1   for (each vertex v ∈ V){
2       pre(v) = NIL;
3       est(v) = ∞;
4   }
5   est(s) = 0;

Relax(Vertex u, Vertex v)
1   if (est(v) > est(u) + w((u, v))){
2       est(v) = est(u) + w((u, v));
3       pre(v) = u;
4   }

- source $s$: the origin of shortest paths
- pre($v$): predecessor of node $v$
- est($v$): shortest-path estimate of $v$, changes along the algorithm

- Relaxation:
  - checks if the shortest path from source $s$ to $v$ can be shortened (relaxed) by taking a path through $u$

# DAG Shortest Path Algorithm

- Perform a topological sort first
- Visit each node in topological order and relax all edges incident from it
- $\Theta(V + E)$ runtime

DAGShortestPaths(Graph G, vertex s)
1  topologically sort the vertices of G;
2  Initialize(G, s);
3  for (each vertex u in topological sorted order)
4      for (each vertex v such that $(u, v) \in E$)
5          Relax(u, v);
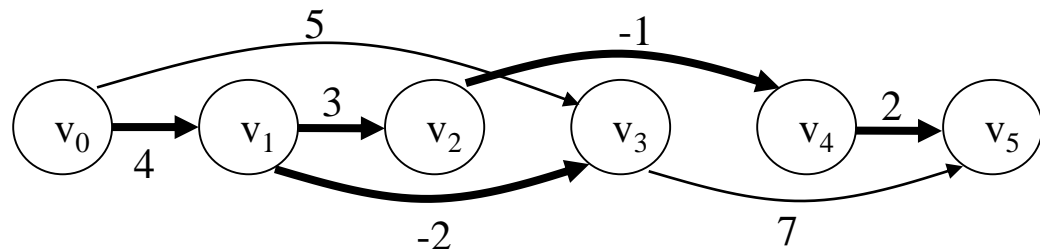
# Dijkstra's Algorithm

- Works on graphs with non-negative weights
- Needs a priority queue *minQ* with est($v$) as keys
- Extracts the node *u* with minimum est($u$) in *minQ* and relaxes all edges incident from *u* to all nodes still in *minQ*

```
Dijkstra(Graph G, Vertex s)
1   Initialize(G, s);
2   Priority_Queue minQ = {all vertices in V};
3   while (minQ ≠ ∅){
4      Vertex u = ExtractMin(minQ); // minimum est(u)
5      for (each v ∈ minQ such that (u, v) ∈ E)
6         Relax(u, v);
7   }
```

# Dijkstra's Algorithm (Cont'd)
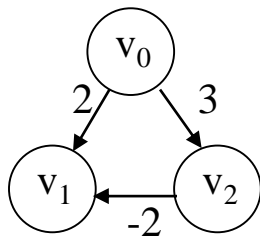


| | Predecessors | | | Shortest-Path Estimates | | |
|---|---|---|---|---|---|---|
| | $v_0$ | $v_1$ | $v_2$ | $v_0$ | $v_1$ | $v_2$ |
| Dijkstra's | NIL | $v_0$ | $v_0$ | 0 | 2 | 3 |
| Correct path | NIL | $v_2$ | $v_0$ | 0 | 1 | 3 |

- Produces incorrect results if weights are negative
- Time complexity depends on the implementation of the priority queue *minQ*
  - A linear array: $O(V^2)$
  - A Fibonacci heap: $O(E + V \cdot \lg V)$

# The Bellman-Ford Algorithm

- Relax every edge ($|V|$ - 1) times
  - Since negative cycles should not exist in a shortest-path problem

- The most general algorithm
  - Also the most time-consuming: $O(VE)$

```
Bellman-Ford(Graph G, Vertex s)
1  Initialize(G, s);
2  for (counter = 1 to |V| - 1)
3     for (each edge (u, v) ∈ E)
4        Relax(u ,v);
5  for (each edge (u, v) ∈ E)
6     if (est(v) > est(u) + w((u, v)))
7        report "negative-weight cycles exist";
```

# The Longest Path Problem

- Similar to the shortest-path problem if no positive-cycle exists
  - Changing the value of est(v) to $-\infty$ in initialization and ">" to "<" in relaxation finds longer paths
- If positive cycles exist and want to find the longest simple path
  - An NP-hard problem

# Minimum Spanning Tree (MST)

- Spanning trees
  - Defined on undirected, weighted graphs
  - Connects all vertices without cycles
  - Tree weight: sum of all edge weights
- Minimum Spanning Tree
  - A spanning tree with minimum tree weight
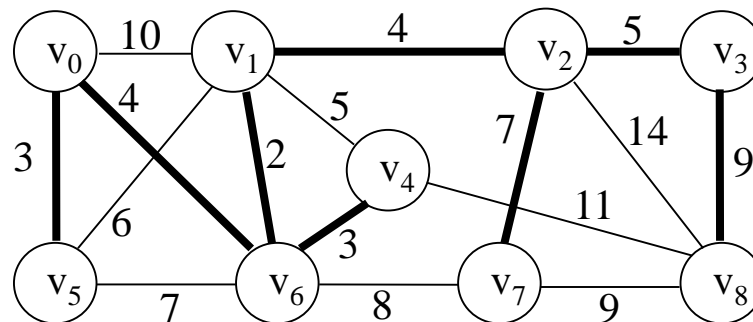
# Prim's Algorithm

- Builds a set with a starting vertex
  - Incrementally adds the shortest edge to the set
  - The set ends with an MST

PrimMST(Graph G)
1  Priority_Queue minQ = {all vertices in V};
2  for (each vertex u ∈ minQ) u.key = ∞;
3  randomly select a vertex r in V as root;
4  r.key = 0;
5  r.predecessor = NIL;
6  while (minQ ≠ ∅){
7    vertex u = ExtractMin(minQ);
8    for (each vertex v such that (u, v) ∈ E)
9      if (v ∈ minQ and w((u, v)) < v.key) {
10        v.predecessor = u;
11        v.key = w((u, v));
12      }
13  }

- Use a priority queue
  - key: edge weight
- Runtime depends on *minQ*
  - Like Dijkstra's Algortihm
  - $O(V^2)$ with an array
  - $O(E + V \lg V)$ with a Fibonacci heap

# Flow Network

- A variant of connected, directed graphs

- Two special nodes
  - Source $s$: no edge incident to $s$
  - Sink $t$: no edge incident from $t$
  - Every flow starts at $s$ and ends at $t$

- Every edge $(u, v)$ has two attributes
  - Capacity $c(u, v)$: the flow it can hold
  - Flow $f(u, v)$ satisfies 3 constraints
    - Capacity constraint: $f(u, v) \leq c(u, v)$
    - Skew symmetry: $f(u, v) = -f(v, u)$
    - Flow conservation (exceptions: $s$ and $t$): 有多少進來，一定都流出去 $\sum_{v \in V} f(u, v) = 0$

# Maximum-Flow Problem

- The value of a flow:  $|f| = \sum_{v \in V} f(s, v)$
- Maximum flow problem
  - Finds a flow with the maximum value in a flow network

- Numbers on edges: $f(u, v)/c(u, v)$
- $|f| = 19$, not maximum
  - More flow can be pushed into path $s \rightarrow v_2 \rightarrow v_3 \rightarrow t$
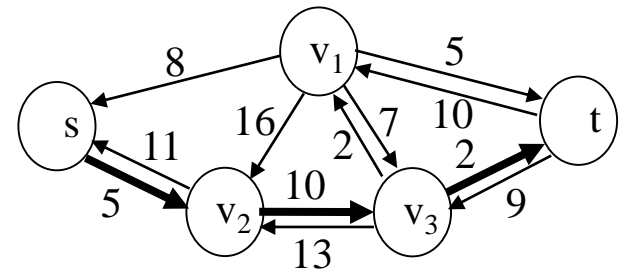    - An augmenting path
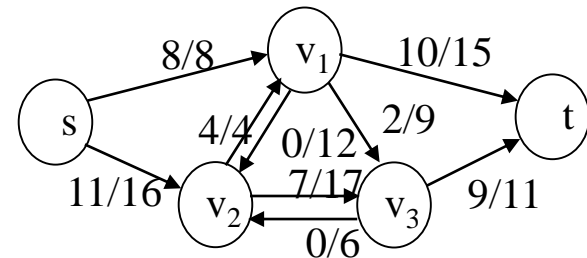
# Residual Network

- Facilitates finding augmenting paths
- Residual capacity
  - Defined with respect to a flow $f$
  - $c_f(u, v) = c(u, v) - f(u, v)$
  - For both directions of every pairs of nodes
- $G_f = (V, E_f)$
  - $E_f$: edges with residual capacity as weights
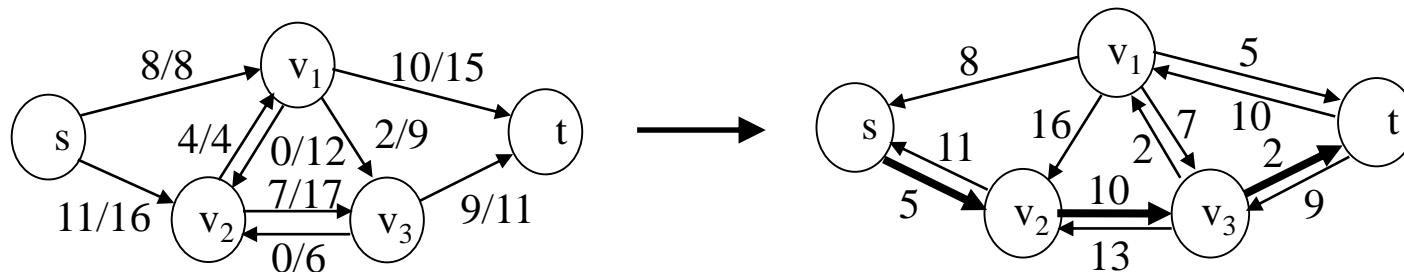
# Residual Network (Cont'd)

- Augmenting paths
  - Paths in the residual network from *s* to *t*
  - E.g., $p = s \rightarrow v_2 \rightarrow v_3 \rightarrow t$
- Residual capacity of a path
  - Minimum edge weight on the path
  - $c_f(p) = c_f(v_3, t) = 2$ <span style="color:orange">代表還可以再增加2個流量</span>
- Intuitive algorithm for maximum flow
  - Finds augmenting paths in residual networks and push flows equal to their residual capacity
  - Updates residual networks according to the new flows until no augmenting paths can be found

# The Ford-Fulkerson Method

- An intuitive method that repeats the following steps
  - Finds an augmenting path $p$ on the residual network
  - Push more flow according to $c_f(p)$
  - Update the residual network

```
Ford-Fulkerson(Graph G, Source s, Sink t)
1   for (each (u, v) ∈ E) f[u, v] = f[v, u] = 0;
2   Build a residual network Gf based on flow f;
3   while (there is an augmenting path p in Gf){
4      cf(p) = min(cf(u, v) : (u, v) ∈ p);
5      for (each edge (u, v) ∈ p){
6         f[u, v] = f[u, v] + cf(p);
7         f[v, u] = -f[u ,v];
8      }
9      Rebuild Gf based on new flow f;
10  }
```

- Time complexity: $O(E \cdot |f^*|)$
  - $f^*$: the maximum flow
  - $|f^*|$ can be very large
  - Very inefficient if $|f^*|$ is large

# The Edmonds-Karp Algorithm

- In Ford-Fulkerson, how to find augmenting paths is unspecified

- Edmonds-Karp algorithm uses breadth-first search to find an augmenting path with a minimum number of edges

- Time complexity: $O(VE^2)$
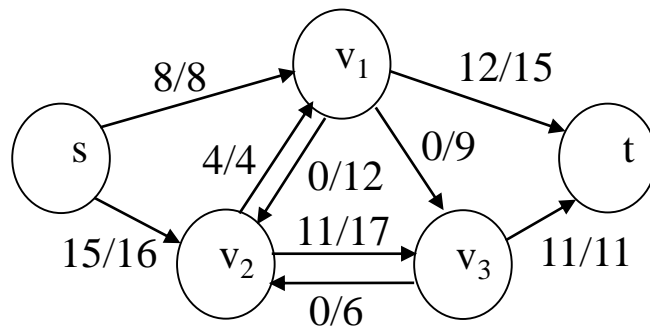


- Resultant network
  - The maximum flow
  - $|f^*| = 23$

# Cuts in Flow Networks

- A cut $(S, T)$
  - A partition of the node set $V$ into $S$ and $T = V - S$
  - Source $s \in S$ and sink $t \in T$
    - $S = \{s, v_2, v_3\}, T = \{t, v_1\}$
  - Net flow across the cut, $f(S, T)$: $f(S,T) = \displaystyle\sum_{u \in S, v \in T} f(u,v)$
    - $f(S, T) = 21$
  - Capacity of the cut, $c(S, T)$: $c(S,T) = \displaystyle\sum_{u \in S, v \in T} c(u,v)$
    - $c(S, T) = 23$
    - $f(S, T) \le c(S, T)$
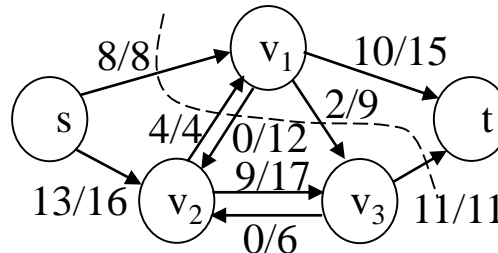
# The Max-Flow Min-Cut Theorem

- The following 3 things are equivalent
  - $f$ is a maximum flow in $G$
  - The residual network $G_f$ has no augmenting paths
  - $|f| = c(S, T)$ for some cut of $G$
- Finding maximum flow = finding minimum cut
  - $|f*| = 23 = c(S, T) =$

    $c(\{s, v_2, v_3\}, \{t, v_1\})$
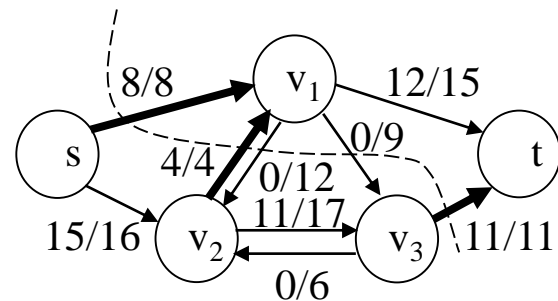
# Maximum Bipartite Matching

- A bipartite graph $G = (V, E)$
  - $V$ is partitioned into two sets $L$ and $R$
  - For every edge $(u, v) \in E$, if $u \in L$, then $v \in R$, and vice versa

- A matching
  - A subset of edges $M \subseteq E$
  - For each vertex in $V$, at most one edge of $M$ is incident on it
  - E.g., 3 thick edges in the bipartite graph
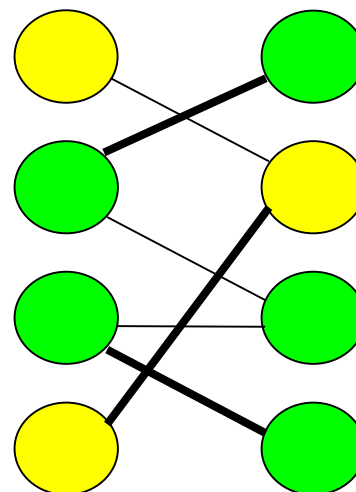
# Maximum Bipartite Matching (Cont'd)

- Maximum bipartite matching finds a matching with a maximum number of edges
  - Add a source $s$ and link $s$ to all nodes in $L$
  - Add a sink $t$ and link all nodes in $R$ to $t$
  - Every edge has unit capacity
  - Solve the maximum flow problem
- Ford-Fulkerson method solves in $O(VE)$
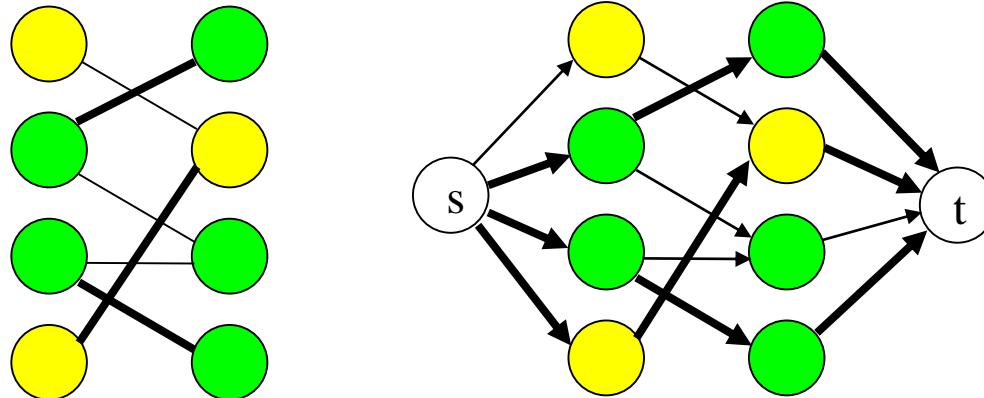
# Heuristic Algorithms

- Applies heuristics, or rules of thumb
- Finds good but not always optimal solutions
- Efficient in time
  - Best for hard (NPC or NP-hard) problems
- Solution quality cannot always be guaranteed
  - Nearest Neighbor for TSP
- Either directly searches the solution space
  - Greedy algorithm, dynamic programming, branch and bound
- Or exerts perturbations on solutions
  - Simulated annealing, genetic algorithms

# Greedy Algorithm

- General idea
  - Stages the optimization problem
  - Makes locally optimal choices at each stage
- Real life example
  - Giving change with a minimum number of coins
  - Heuristic: pick the coin with the greatest value
  - 36 cents: quarter → dime → penny: 3 coins
- Two properties make greedy algorithms work
  - Greedy choice
  - Optimal substructure
- Applications: Dijkstra's, Prim's algorithms

# Greedy Choice Property

- The global optimal solution can be made by making locally optimal, or greedy, choices

- Does not consider the impact of the current choice on future choices

- Counterexamples
  - Nearest Neighbor for TSP
  - Giving change of 40 cents if there were 20-cent coins
    - (Greedy) quarter → dime → nickel: 3 coins
    - (Optimal) 2 20-cent coins: 2 coins

# Optimal Substructure Property

- The global optimal solution consists of optimal solutions to its subproblems
    - The problem is divisible into subproblems
    - The combination of optimal solutions to subproblems is globally optimal
- Giving change of 36 cents
    - into 26 cents + 10 cents:
    - (quarter → penny) + dime : global optimal

# Dynamic Programming (DP)

- Combines solutions to its dependent subproblems by utilizing the dependency
  - Unlike divide-and-conquer: subproblems are independent
  - Avoids repeatedly solving the same subproblems
- Example: matrix-chain multiplication
  - Find the multiplication sequence with the least number of scalar multiplications
  - Matrices A, B, C: 30 x 100, 100 x 2, 2 x 50
    - (AB)C: #scalar multiplications = 9000
    - A(BC): #scalar multiplications = 160,000

# Matrix-Chain Multiplication

- A chain of $n$ matrices $<M_1, M_2, \ldots, M_n>$
  - $M_i$: dimension $v_{i-1}$ x $v_i$
- Search all possible multiplication orders
  - Exponential with $n$: infeasible
- $m[i, j]$: minimum number of scalar multiplications for $M_iM_{i+1}\ldots M_j$
  - Recurrence relation:

$$m[i, j] = \begin{cases} 0 & if\ i = j, \\ \min_{i \le k < j}\{m[i,k] + m[k+1, j] + v_{i-1}v_kv_j\} & if\ i < j. \end{cases}$$

  - Target: find $m[1, n]$

# Matrix-Chain Multiplication (Cont'd)

BottomUpMatrixChain(Vector v)
1   n = v.size – 1;
2   for (i = 1 to n) m[i, i] = 0;
3   for (p = 2 to n) // p: chain length
4       for (i = 1 to n – p + 1){
5           j = i + p – 1;
6           m[i, j] = ∞;
7           for (k = i to j - 1){
8               temp = m[i, k] + m[k + 1, j]
                    + $v_{i-1}v_k v_j$;
9               if (temp < m[i, j]){
10                  m[i, j] = temp;
11                  d[i, j] = k;
12              }
13          }
14      }
15  return m and d;

- $d[i, j]$
  - where the separation is
- Constructs the $m[i,j]$, $(0 \leq i \leq j \leq n)$ table in a bottom-up fashion
- Avoids repeated calculations of $m[i, j]$ in a naive recursive function
- Time complexity: $O(n^3)$

# Two Properties for DP

- Overlapping Subproblems
  - The decomposed subproblems are dependent or overlapped
- Optimal Substructure
  - The same as in greedy algorithms
  - DP or greedy?
    - Whether the problem has "overlapping subproblems" or "greedy choice"
- Matrix-chain multiplication has both

# Memoization

- Constructs the table in a top-down, recursive fashion

TopDownMatrixChain(Vector v)
1   n = v.size – 1;
2   for (i = 1 to n)
3     for (j = i to n) m[i, j] = ∞;
4   return Memoize(v, 1, n);

Memoize(Vector v, Index i, Index j)
1   if (m[i ,j] < ∞) return m[i, j];
2   if (i = j) m[i, j] = 0;
3   else for (k = i to j - 1){
4       temp = Memoize(v, i, k) +
5           Memoize(v, k + 1, j) + $v_{i-1}v_kv_j$;
6       if (temp < m[i, j]) m[i, j] = temp;
7     }
8   return m[i, j];

- "Memo"ization
  - Takes "memo"s
  - Records table items along the recursion
- Sometimes better than the bottom-up approach
  - If not all table entries must be visited

# Branch and Bound

- Branching
  - Makes several choices at the same point to branch out into the search space
  - The solution space forms a tree-like structure
  - Fully-branched space: too vast to explore
- Bounding and pruning
  - Estimates a lower bound on solution quality to prune out obviously impossible branches
  - Efficiently reduces the solution space made with branching

# Branch and Bound for TSP

- Branching: the next node on the route
- Bounding: use MST to estimate the cost lower bound of unvisited route
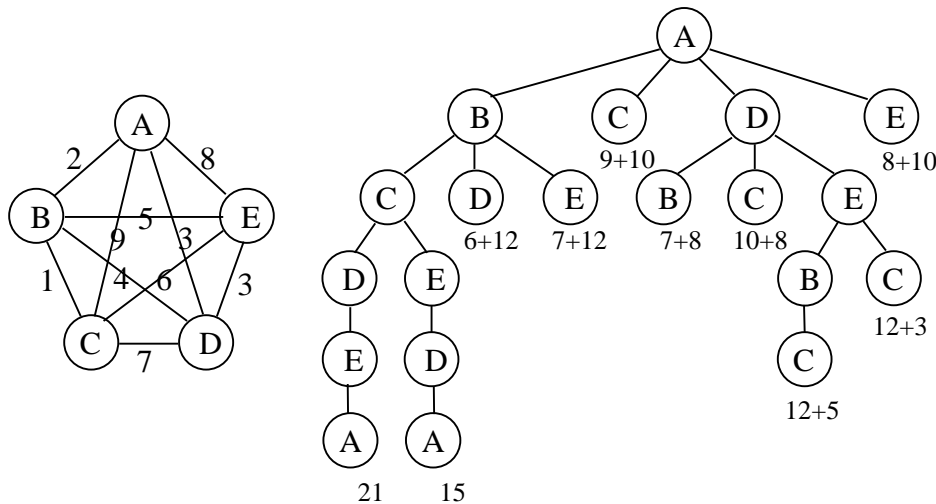
# Simulated Annealing (SA)

- Mimics the process of controlled cooling
  - The heat gives molecules huge energy first
    - Large perturbations
  - Slow cooling gradually deprives the energy
    - Chances to reach stable crystalline configuration
- Analogies
  - Energy $\rightarrow$ cost function
  - Movements of molecules $\rightarrow$ perturbations in the solution space
  - Temperature $\rightarrow$ control parameter $T$

# Pseudocode for SA

Accept(temperature T, cost $\Delta c$)
1   Choose a random number "rand" between 0 and 1;
2   return ($e^{-\Delta c/T}$ > rand);

SimulatedAnnealing()
1   solution sNow, sNext, sBest;
2   temperature T, endingT;
3   Initialize sNow, T and endingT;
4   while (T > endingT){
5     while(!ThermalEquilibrium(T)){
6       sNext = Perturb(sNow);
7       if (cost(sNext) < cost(sNow)){
8         sNow = sNext;
9         if (cost(sNow) < cost(sBest))
10            sBest = sNow;
11        }
12      else if (Accept(T, cost(sNext)-cost(sNow)))
13        sNow = sNext;
14    }
15    Decrease(T);
16  }
17  return sBest;

- Cooling schedule
  - Combination of ThermalEquilibrium, Decrease, and endingT
  - Characterization of the SA

- Advantages
  - Can escape local optima
  - Flexible in running time

# Genetic Algorithms (GA)

- Inspired by evolutional biology
  - Uses operations like crossover and mutation
  - Operates on a set of feasible solutions
    - "population"
  - Solutions encoded in math symbols like bits
    - "genes"
  - A feasible solution is a bit string
    - a "chromosome"
  - "Survival of the fittest"
    - Solutions are evaluated by a "fitness function"

# Simple Genetic Algorithms (SGA)
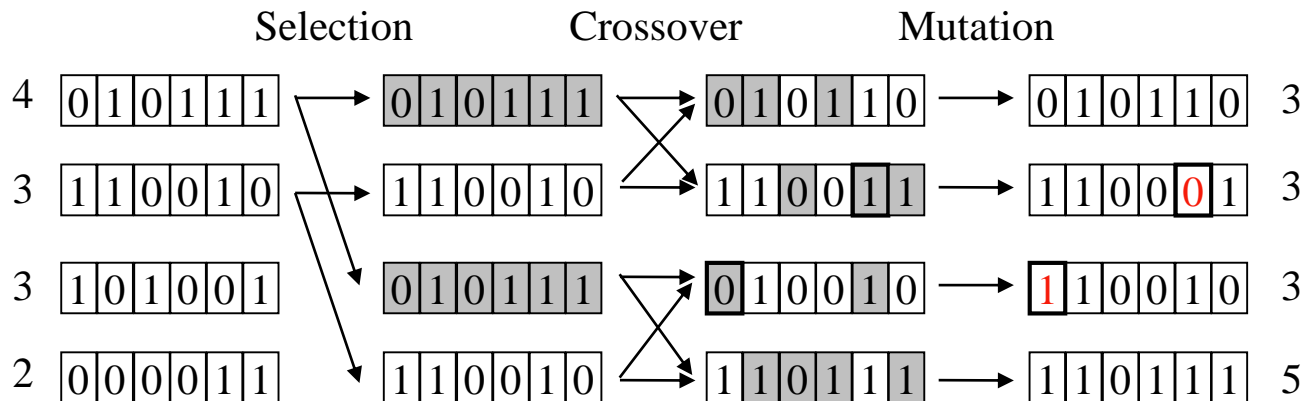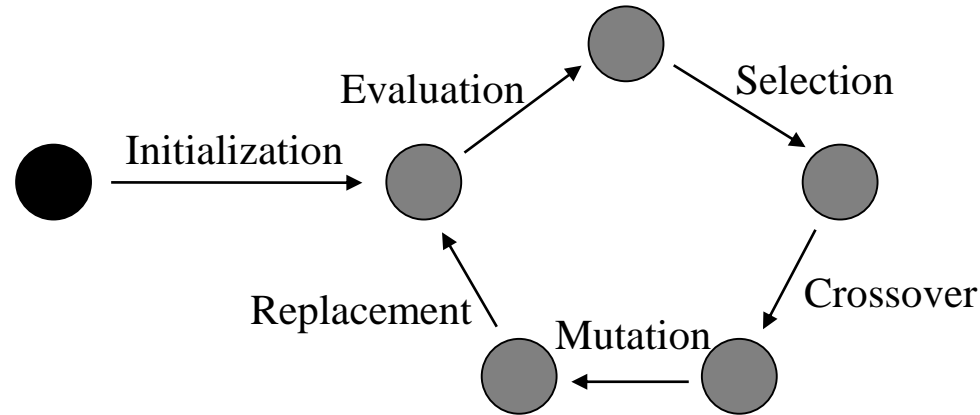
Average fitness: **3**
Highest fitness: **4**

Average fitness: **3.5**
Highest fitness: **5**

# Mathematical Programming

- Problem formulation

  ⇨ Minimize (or maximize) $f(x)$;

  ⇨ Subject to $X = \{ \ x \mid g_i(x) \leq b_i, \ i = 1...m \ \}$;

  where

  - $x = (x_1,..., x_n)$ are optimization (or decision) variables,

  - $f : R^n \rightarrow R$ is the objective function

  - $g_i : R^n \rightarrow R$ and $b_i \in R$ form the constraints for the valid values of $x$.

# Categories of Mathematical Programming Problems

1. If $X = R^n$, the problem is unconstrained.
2. If $f$ and all the constraints are linear, the problem is called a linear programming (LP) problem.
   - Constraints can then be represented in the matrix form $Ax \leq B$, where $A$ is an $m \times n$ matrix corresponding to the coefficients in $g_i(x)$.
3. If the problem is linear, and all the variables are constrained to integers, the problem is called an integer linear programming (ILP) problem.
   - If only some of the variables are integers, it is called a mixed integer linear programming (MILP) problem.

# Categories of Mathematical Programming Problems (Cont'd)

4. If the constraints are linear, but the objective function $f$ contains some quadratic terms, the problem is called a quadratic programming (QP) problem.

5. If $f$ or any of $g_i(x)$ is not linear, it is called a nonlinear programming (NLP) problem

6. If all the constraints have the following convexity property: $g_i(\alpha x_a + \beta x_b) \leq \alpha\, g_i(x_a) + \beta\, g_i(x_b)$

   where $\alpha \geq 0$, $\beta \geq 0$, and $\alpha + \beta = 1$

   then the problem is called a convex programming or convex optimization problem.

7. If the set of feasible solutions defined by $f$ and $X$ are discrete, the problem is called a discrete or combinatorial optimization problem.
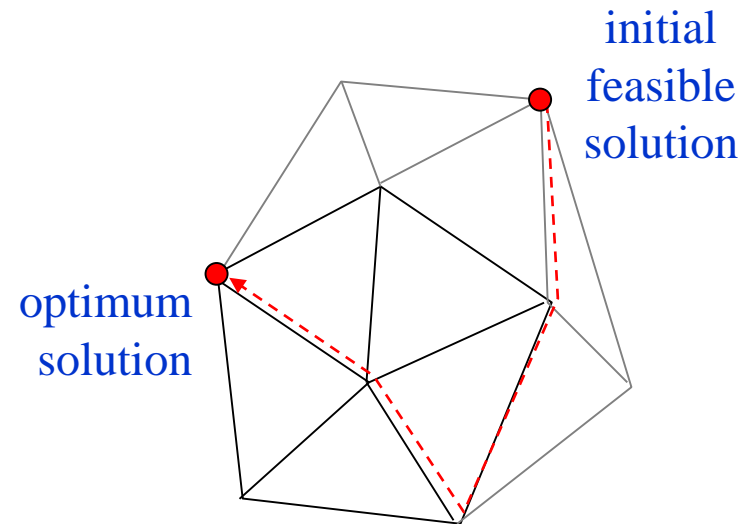
# Linear Programming (LP) Problem

- Intuition: solving LP problems should be simpler than solving the general mathematical optimization problems

- Fact: A polynomial-time algorithm was not available until 1970's

# LP Formulation

- One linear equation forms a hyper-plane.

- One linear inequality (constraint) forms a hyper half-space.

- The set of linear constraints forms a polyhedron in a multi-dimensional space.

- The optimal value of a linear objective function over a set of linear constraints occurs at an extreme point of the polyhedron.

# Simplex Method

- Developed by George Dantzig in 1947
  - First practical procedure used to solve the LP problem
- Finds a basic feasible solution that satisfies all the constraints
  - A basic solution is conceptually a *vertex* (i.e., an extreme point) of the convex polyhedron
- Moves along the edges of the polyhedron in the direction towards finding a better value of the objective function
  - Guaranteed to eventually terminate at the optimal solution

initial feasible solution

optimum solution

# Integer Linear Programming (ILP) Problem

- Fact:
  - Many EDA problems are best formulated with integer variables
    - E.g., signal values in a digital circuit are under a modular number system
    - E.g., problems that need to enumerate the possible cases, or are related to scheduling of certain events
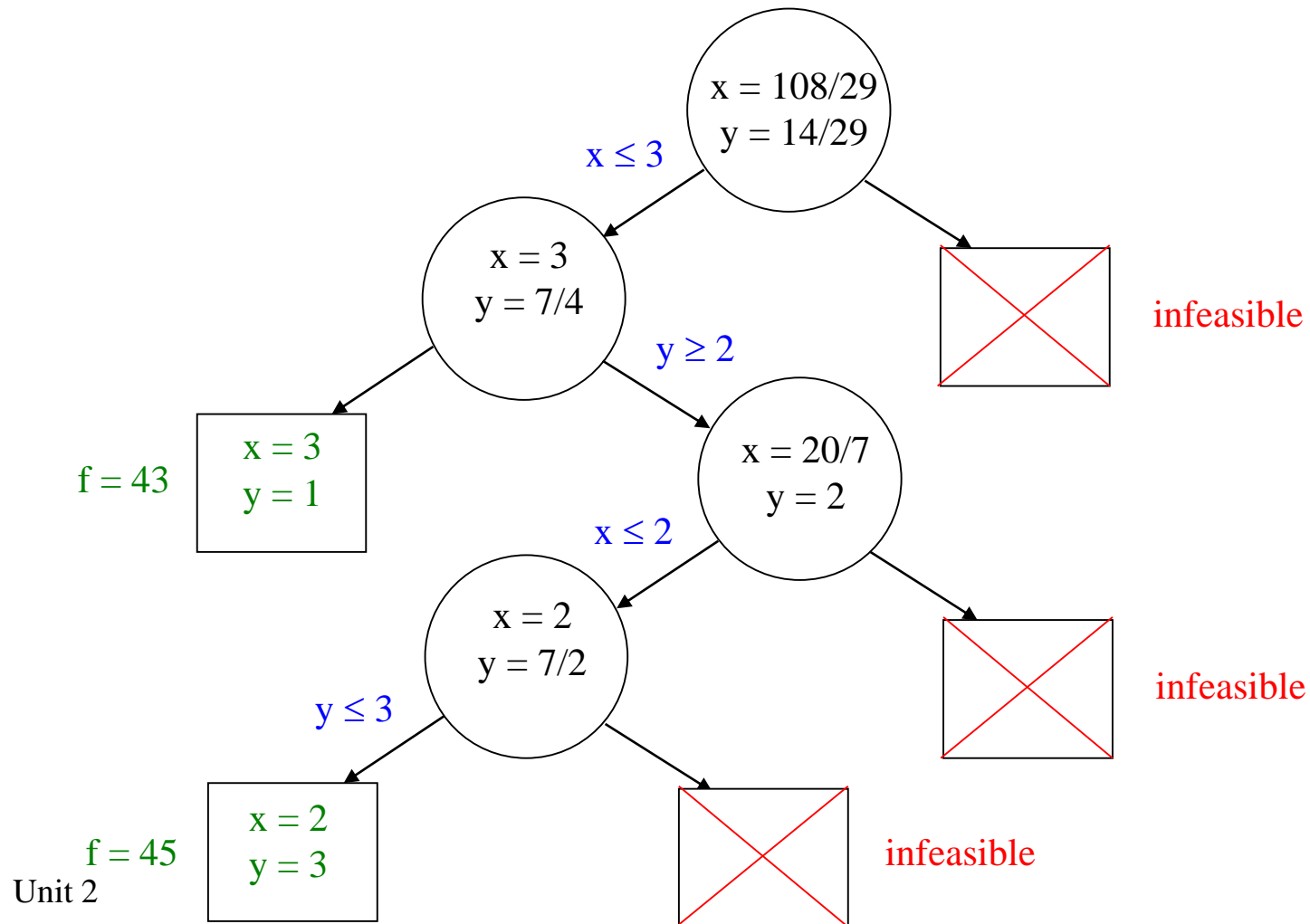  - In general more difficult than the LP counterpart

# An ILP Example

- maximize
  - $f$: $12x + 7y$
- subject to
  - $g_1$: $2x - 3y \leq 6$
  - $g_2$: $7x + 4y \leq 28$
  - $g_3$: $-x + y \leq 2$
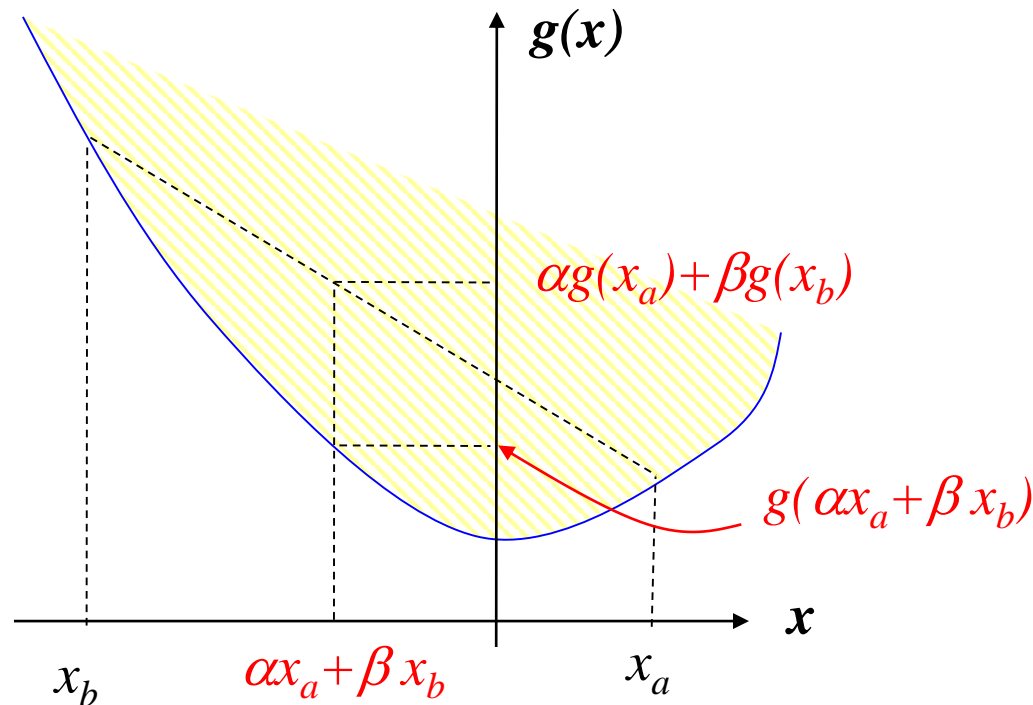  - $g_4$: $-2x - y \leq 2$

  where x, y $\in$ Z
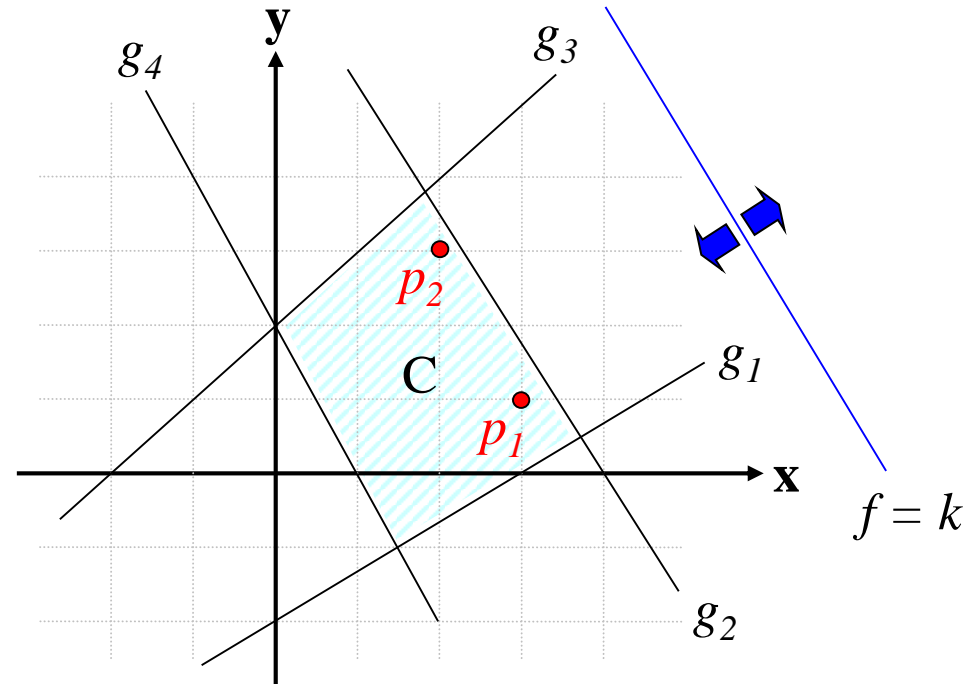


*$p_1$ and $p_2$ are two possible points for optimal solution*

# LP Relaxation and Branch-and-Bound Procedure



$x = 108/29$
$y = 14/29$

$x \le 3$

$x = 3$
$y = 7/4$

infeasible

$y \ge 2$

$f = 43$

$x = 3$
$y = 1$

$x = 20/7$
$y = 2$

$x \le 2$

$x = 2$
$y = 7/2$

infeasible

$y \le 3$

$f = 45$

$x = 2$
$y = 3$

infeasible

# Convex Optimization Problem

- Convexity property
  - $g_i(\alpha x_a + \beta x_b) \leq \alpha \, g_i(x_a) + \beta \, g_i(x_b)$
  - For a convex function, a local optimal solution is also a global optimal solution

# Interior-Point Method

- An effective method that can solve the convex optimization problem in polynomial time within a reasonably small number of iterations

- Idea
  - Obtains an initial feasible solution which is approximated as an interior point
  - Iterates along a path, called a central path, as the approximation improves towards the optimal solution