Learn DAA : From B K Sharma

# TCS-503: Design and Analysis of Algorithms

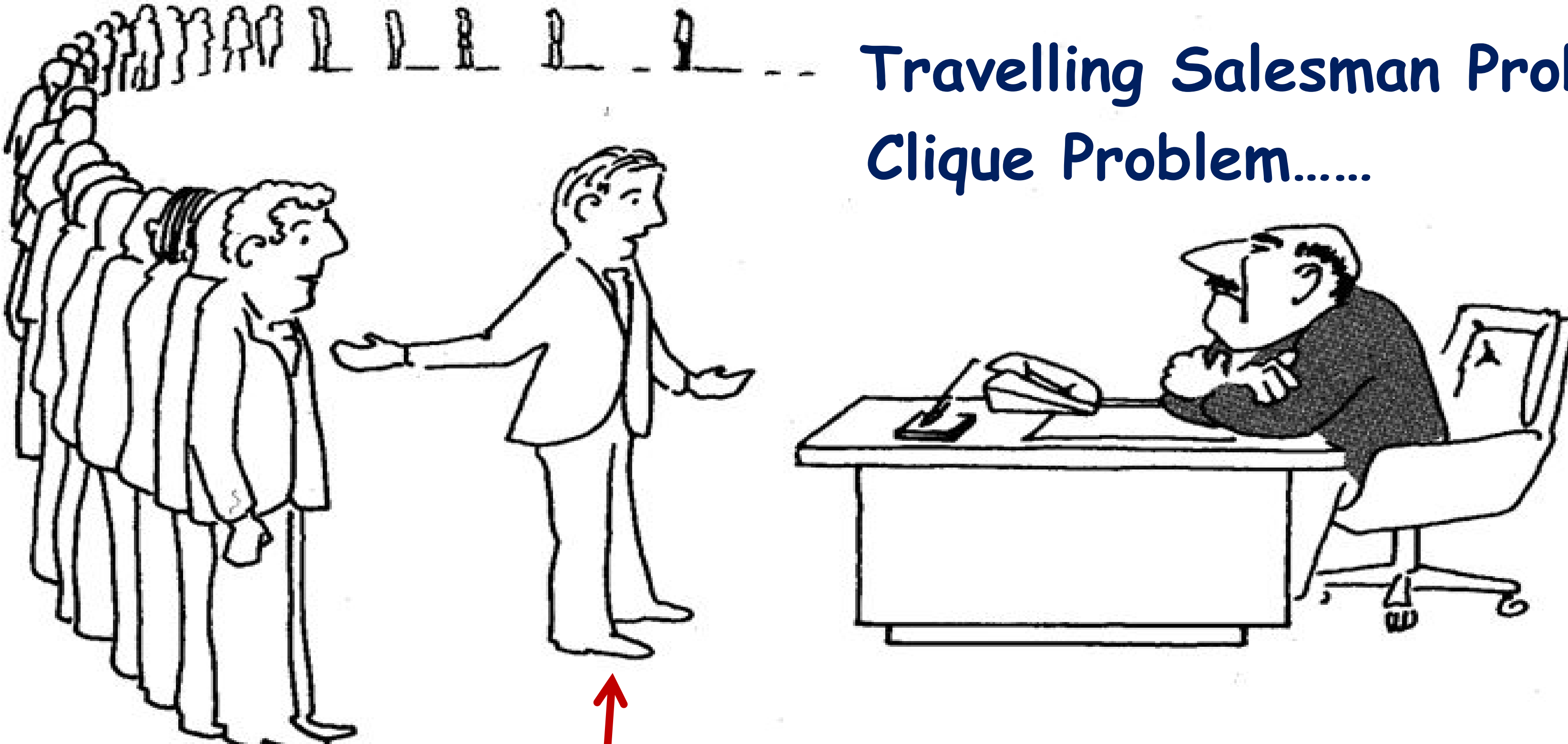
## Unit V: Selected Topics: NP Completeness

# Unit V

- Selected Topics:
  - String Matching
  - NP Completeness
  - Randomized Algorithms
  - Approximation Algorithms

# *NP-completeness*

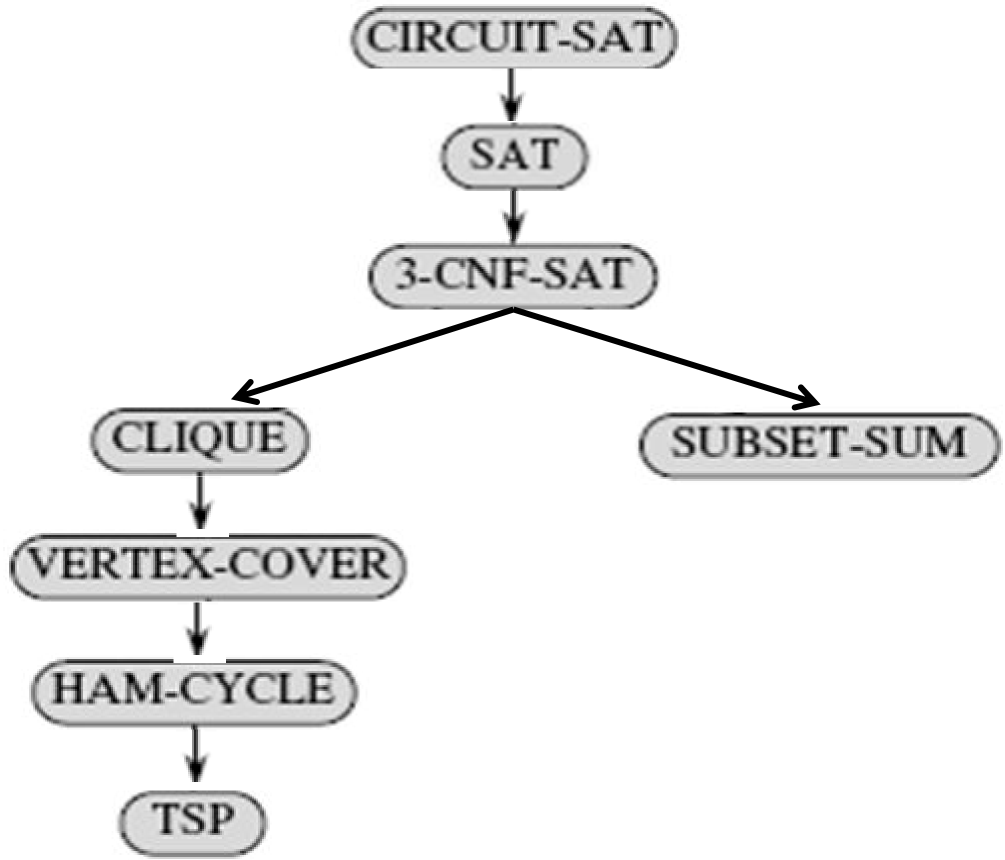
Travelling Salesman Problem,
Clique Problem......



I can't find an efficient algorithm for,

But neither can these famous people

# NP-completeness



# Polynomial Time Algorithms

**Time Complexity:** $O(n^k)$ , $O(n^k.m^k)$ , $O(n^k.\log(n^k))$

Where,

k is a constant, k=1,2,3.......

| | | | | |
|---|---|---|---|---|
| $n =$ | 2, | 10 | 20 | 30 |
| $n^k=$ | $2^1$ | $10^2$ | $20^3$ | $30^4$ |
| $=$ | 2 | 100 | 8000 | 810000 |

# Non-Polynomial Time Algorithms /Exponential time Algorithms

**Time Complexity:** $O(2^n)$ , $O(n.2^n)$, $O(n!)$ , $O(n^n)$

| | | | | |
|---|---|---|---|---|
| $n=$ | 2 | 10 | 20 | 30 |
| $2^n=$ | 4 | 1024 | 1 million | 1000 million |

# Introduction

**Almost all the algorithms we have studied thus so far have been polynomial-time algorithms:**

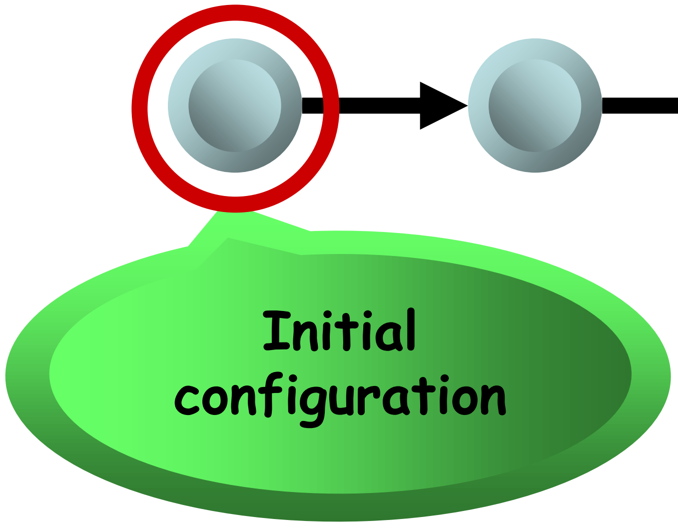
- Bubble Sort, Selection Sort: $O(n^2)$ worst case
- Insertion Sort: $O(n^2)$ worst case
- Quick Sort: $O(n^2)$ worst case
- Counting Sort, Radix Sort, Bucket Sort: $O(n)$
- Other Algorithms: $O(nm^2)$, $O(n\lg n)$

**Non-polynomial Time Algorithms:**

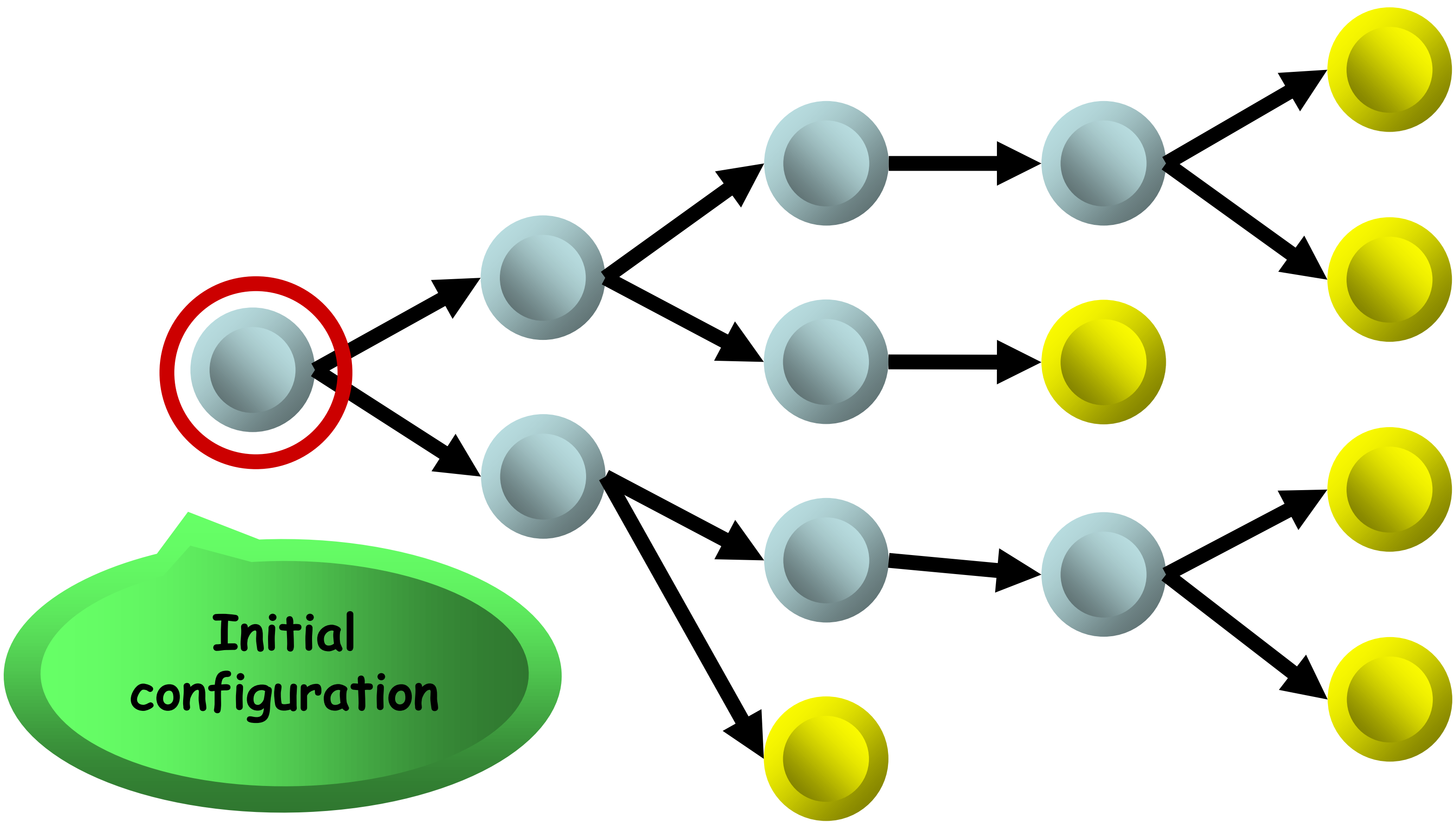
- Travelling Salesman Problem: $O(n\ 2^n)$
- Clique Problem: $O(n\ 2^n)$

Learn DAA : From B K Sharma

## Deterministic algorithm

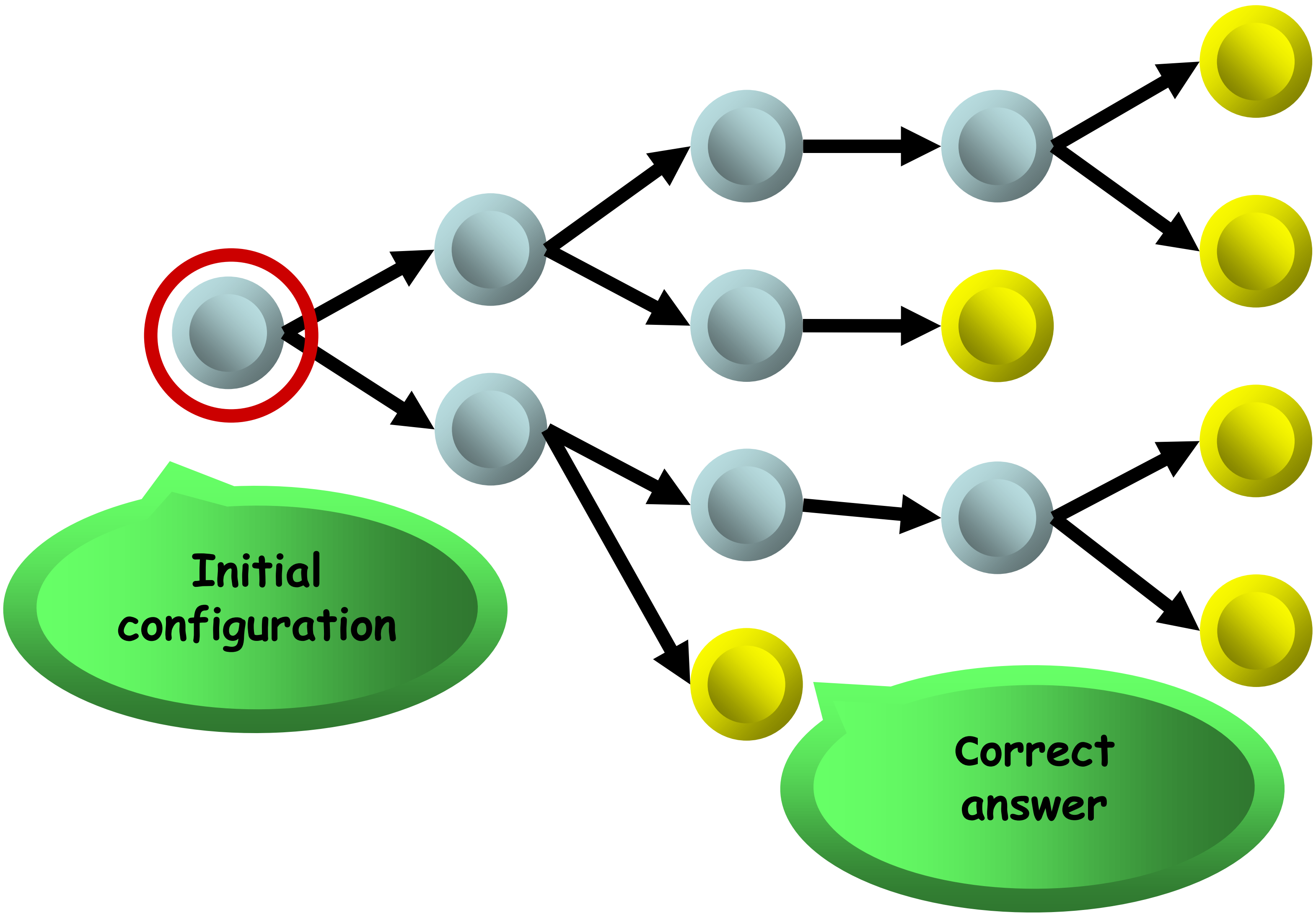


## Non-Deterministic algorithm



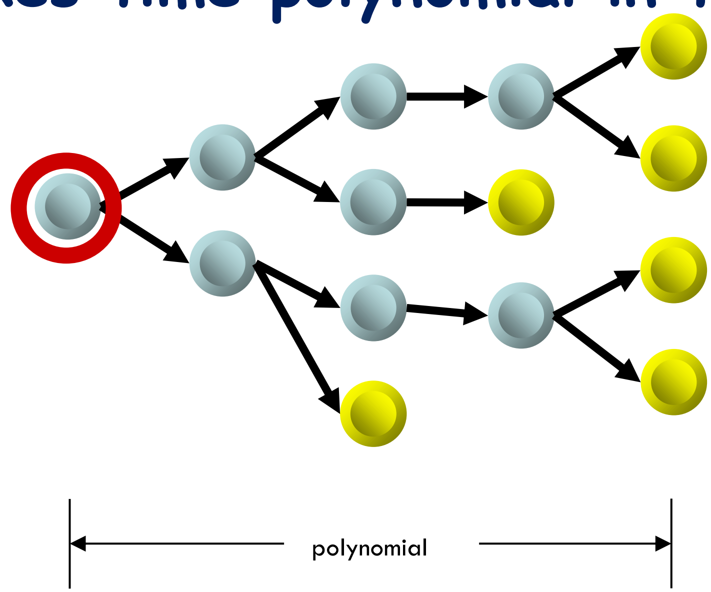
Learn DAA : From B K Sharma

## Non-Deterministically Solving a problem



## Non-Deterministic polynomial time algorithm

We say that a non-deterministic algorithm N runs in polynomial time if for any input x of N, any computation of N on x, takes time polynomial in the size of x.



Learn DAA : From B K Sharma

# Four Classes of Problems

- P — Polynomial Time
- NP — Non-Deterministic Polynomial Time
- NPH — NP- Hard
- NPC — NP-Complete

# Class P

The class P consists of all decision problems that can be solved in polynomial time $O(n^k)$, by **deterministic**, **computers**(the ones that we have used all our life!).

For examples:

Adding two numbers: $O(1)$ "constant"

Looking for an element in an array: $O(N)$ "lineal"

Extracting the element with the highest priority from a heap:

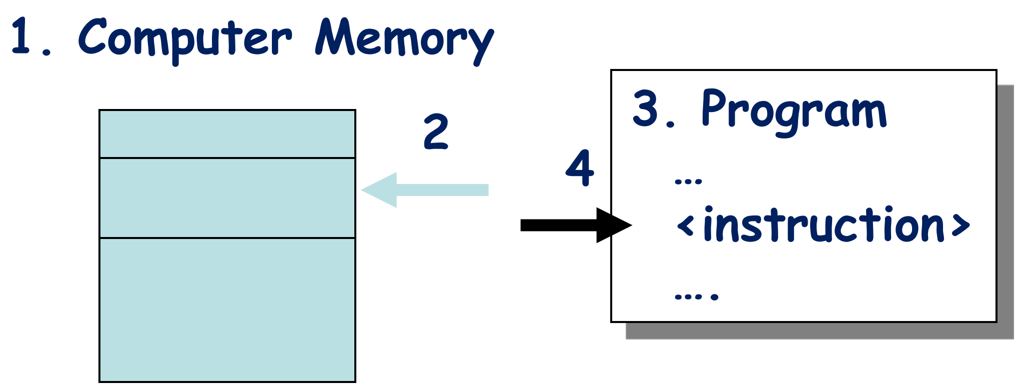
$O(\log N)$ "logarithmic" (thus, $O(N)$ because $N \geq \log N$ )

Looking for the MST:

$O(N \log N)$ (thus, $O(N^2)$ )

# What does Deterministic Computer Means? (Idea)

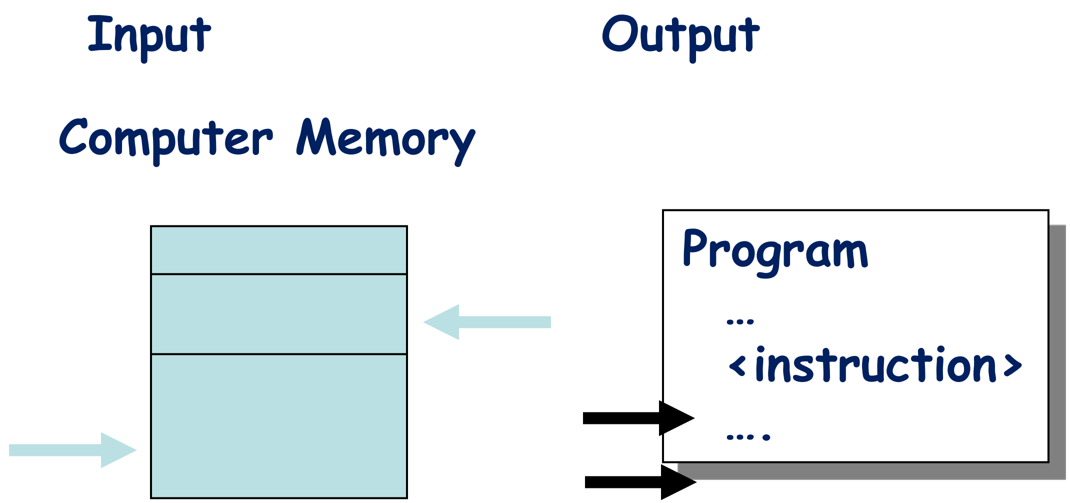
At every computational cycle we have the so-called **state of the computation**:




State = (1. memory, 2. location of memory being pointed at, 3. program, 4. current instruction)

# What does Deterministic Computer Means? (Idea II)

In a deterministic computer we can determine in advance for every computational cycle, the output state by looking at the input state.



# What does Deterministic Computer Mean? (Example)

//input: an array A[1..N] and an element el that is in the array

//output: the position of el in A

```
search(el, A, i)
  {
    if (A[i] = el) then return i
    else
      return search(el, A, i+1)
  }
```

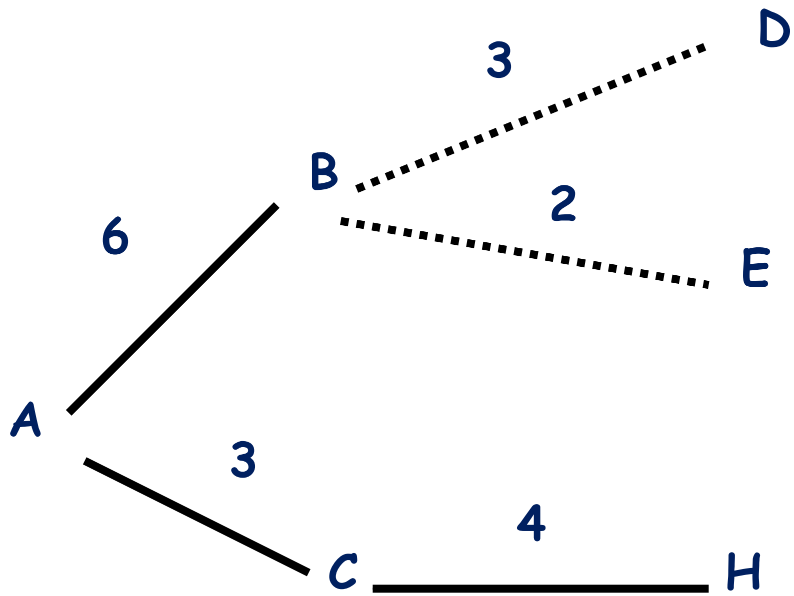
el = 9

7 5 3 8 9 3

Complexity: O(N)

# Djikstra's Shortest Path Algorithm

If the source is A, which edge is selected in the next iteration?




Complexity: O(N log N) ( N = number of edges + vertices)

# Class P

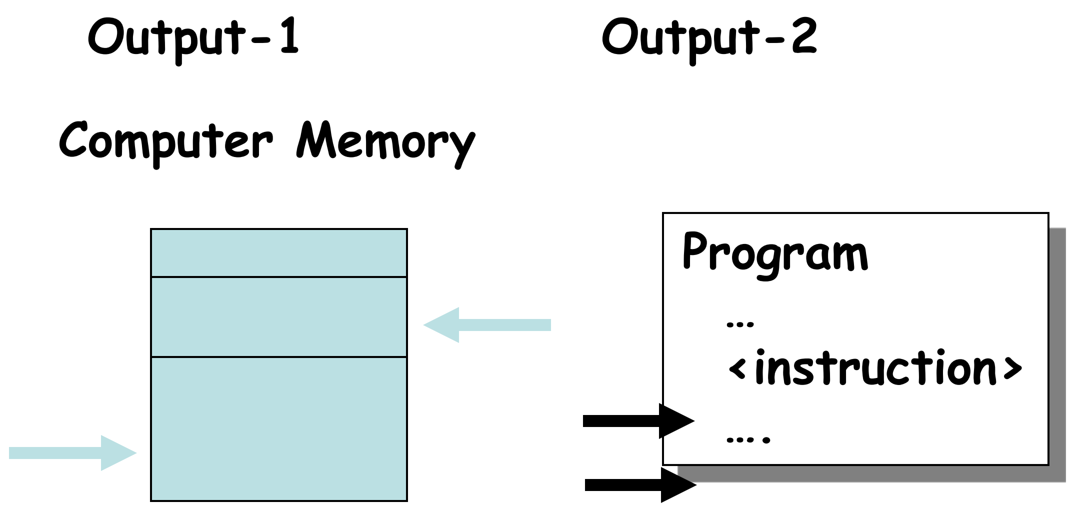
## Formal Definition

*Class P* is a class of decision problems that can be solved in polynomial time by (deterministic) algorithms.

This class of problems is called polynomial.

# What does Non-Deterministic Computer Mean?

In a non-deterministic computer, we may have more than one output state.




The computer "chooses" the correct one ("nondeterministic choice")

# *Nondeterministic algorithm*

## Formal Definition

*A nondeterministic algorithm* is a two-phase procedure that takes as its input an instance *I* of a decision problem and does the following.

**Phase I:** Non-deterministic ("guess") phase:
Generate a candidate solution *S* to the given instance *I*

**Phase II:** Deterministic ("verification") Phase:
A deterministic algorithm takes both *I* and *S* as its input, and it outputs yes if *S* is a solution to instance *I*.

# Non-Deterministic Algorithm

## Formal Definition

A nondeterministic algorithm for a problem X is a two-stage procedure:

- In the first phase, a procedure makes a guess about the possible solution for X.
- In the second phase, a procedure checks if the guessed solution is indeed a solution for X.

```
Phase1(el, A)
{
    i ← random(1..N)
     return i
}
```

```
Phase2(i,el, A)
{
    return A[i] == el
}
```

Note: the actual solution must be included among the possible guesses of phase 1

# Class NP

## Formal Definition

*Class NP* is the class of decision problems that can be solved by nondeterministic polynomial algorithms.

This class of problems is called nondeterministic polynomial.

Contains Problems that are verifiable in polynomial time.

Given a "certificate" of a solution, we can verify that the solution is correct in polynomial time.

# Class NP

The class NP consists of all problems that can be solved in polynomial time by nondeterministic algorithms. (that is, both phase 1 and phase 2 run in polynomial time).

If X is a problem in P then X is a problem in NP because

Phase 1: use the polynomial algorithm that solves X

Phase 2: write a constant time procedure that always returns true.

# NP Class

How to proof that a problem $X$ is in NP:

1. Show that $X$ is in P, or
2. Write a nondeterministic algorithm solving X that runs in polynomial time.

# NP Class

## Showing that searching for an element in an array is in P:

1. Write the procedure search(el, A, i) which runs in lineal time,

//input: an array A[1..N] and an element el that is in the array

//output: the position of el in A

```
search(el, A, i)
  {
    if (A[i] = el) then return i
    else
      return search(el, A, i+1)
   }
```

el = 9

7 5 3 8 9 3

Complexity: O(N)

# Class NP

## Showing that searching for an element in an array is in P:

OR

## 2. Write a non-deterministic algorithm solving search

```
Phase1(el, A)
{
    i ← random(1..N)
    return i
}
```

```
Phase2(i,el, A)
{
    return A[i] == el
}
```

(both Phase 1 and Phase 2 run in constant time)

# Class NP

There is a large number of important problems, for which no polynomial-time algorithm has been found, nor the impossibility of such an algorithm has been proved.

Some samples are:

1. Knapsack Problem
2. Traveling Salesman Problem
3. Graph Coloring Problem
4. 3-CNF-SAT Problem

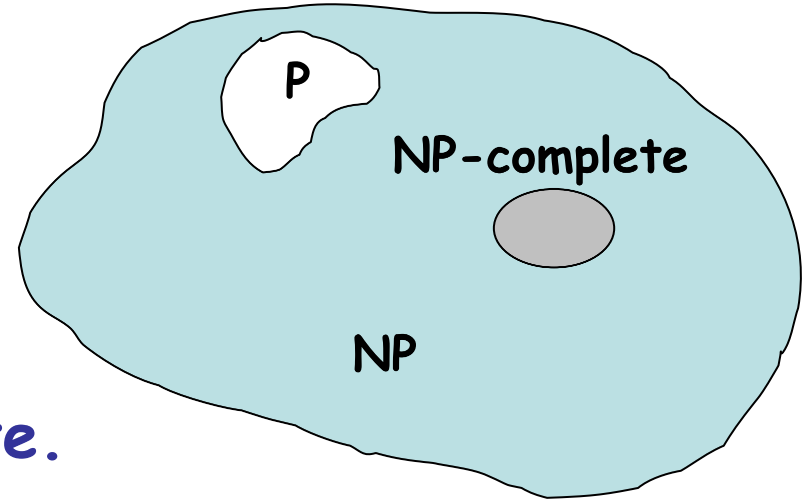
# Is P=NP?

The key question is:

are there problems in NP that are not in P or is P = NP?

Any problem in P is also in NP:

$$P \subseteq NP$$




We can solve problems in P,

even without having a certificate.

The big (and open question) is whether $NP \subseteq P$ or $P = NP$

i.e., if it is always easy to check a solution,

should it also be easy to find a solution?

# Is P=NP?

Most computer scientists believe that this is false but we do not have a proof …

We know that $P \subseteq NP$ since a deterministic TM is also a nondeterministic TM.

But it is unknown if P = NP.

The Clay Mathematics Institute has offered a million dollar prize to anyone that can prove that P=NP or that $P \neq NP$.

P is clearly a subset of NP.

**Theorem:** If any NP-Complete problem can be solved in polynomial time $\Rightarrow$ then P = NP.

# A Decision Problem 'A' Polynomially Reducible To A Decision Problem 'B'

$A \leq_p B$

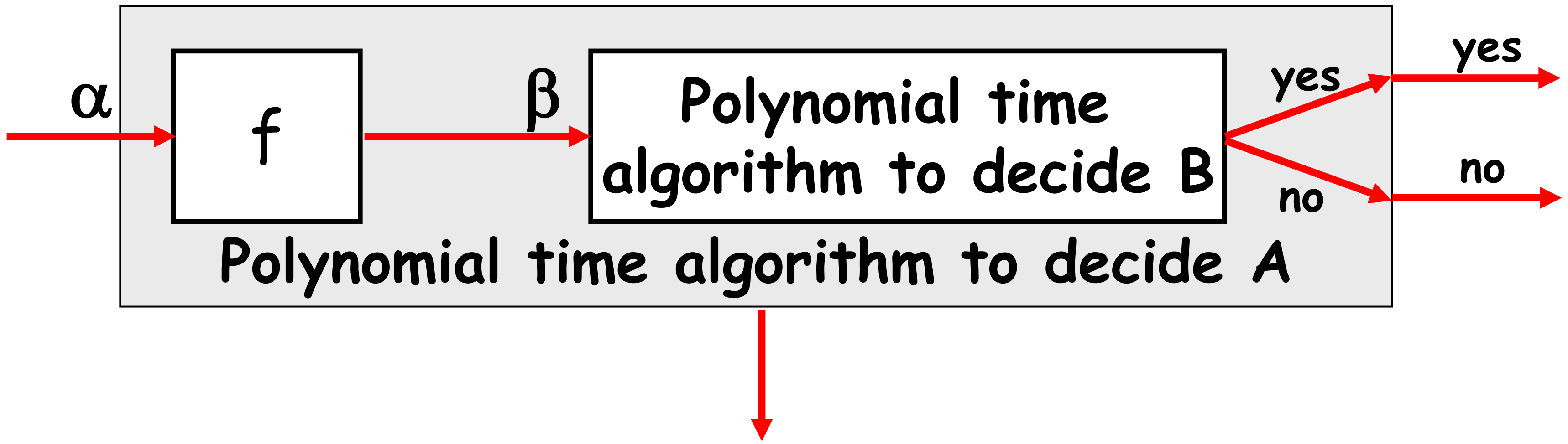
## Formal Definition

A decision problem 'A' is said to be *polynomially reducible* to a decision problem 'B' if there exists a function 'f' that transforms instances of A ($\alpha$) to instances of B ($\beta$) such that

1. 'f' maps all 'yes' instances of A to 'yes' instances of B and 'no' instances of A to 'no' instances of B
2. 'f' is computable by a polynomial-time algorithm.

# Polynomially Reducible To B Decision Problem

## $A \leq_p B$




If a problem A polynomially reducible to some problem B that can be solved in polynomial time, then problem A can also be solved in polynomial time.

# Class NP-H(Hard)

A problem B is NP-hard if every problem ("complete set") in NP can be reduced to B in polynomial the time.

Every problem in *NP* is polynomially reducible to B.

All NP problems $P_1, P_2, P_3, P_4 \ldots \leq_p B$

$P_1$

$P_2$

$P_3$

$P_4$

$\vdots$

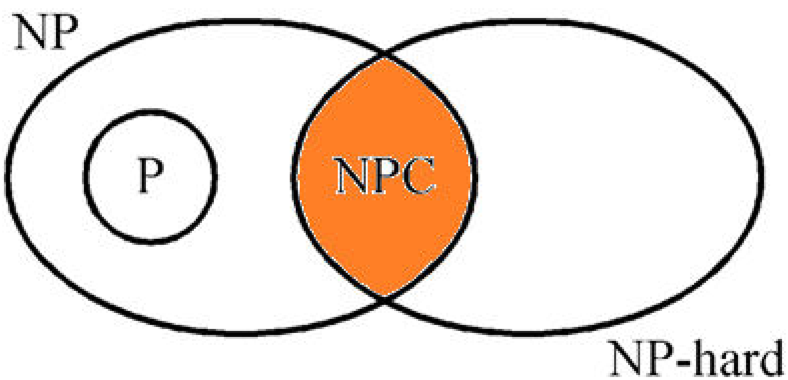
All NP Problems $\leq_p$ B

(B is the hardest problem in the set NP)

# Class *NP-C*

A decision problem *X* is said to be ***NP-complete*** if

1. It belongs to class *NP* and
2. It is NP-hard



# Class NP-C

None of NPC problem has polynomial time algorithm.

So unlikely to find efficient algorithms.

One way to get around NP-completeness:

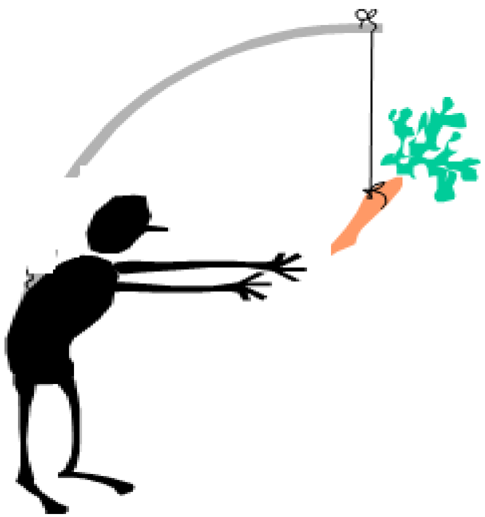
Find *near-optimal solutions* *in polynomial time*.

Is Close Enough Good Enough?

# Approximation Algorithms

An algorithm that returns an answer *C* which is "close" to the optimal solution *C** in polynomial time is called an *approximation algorithm*.

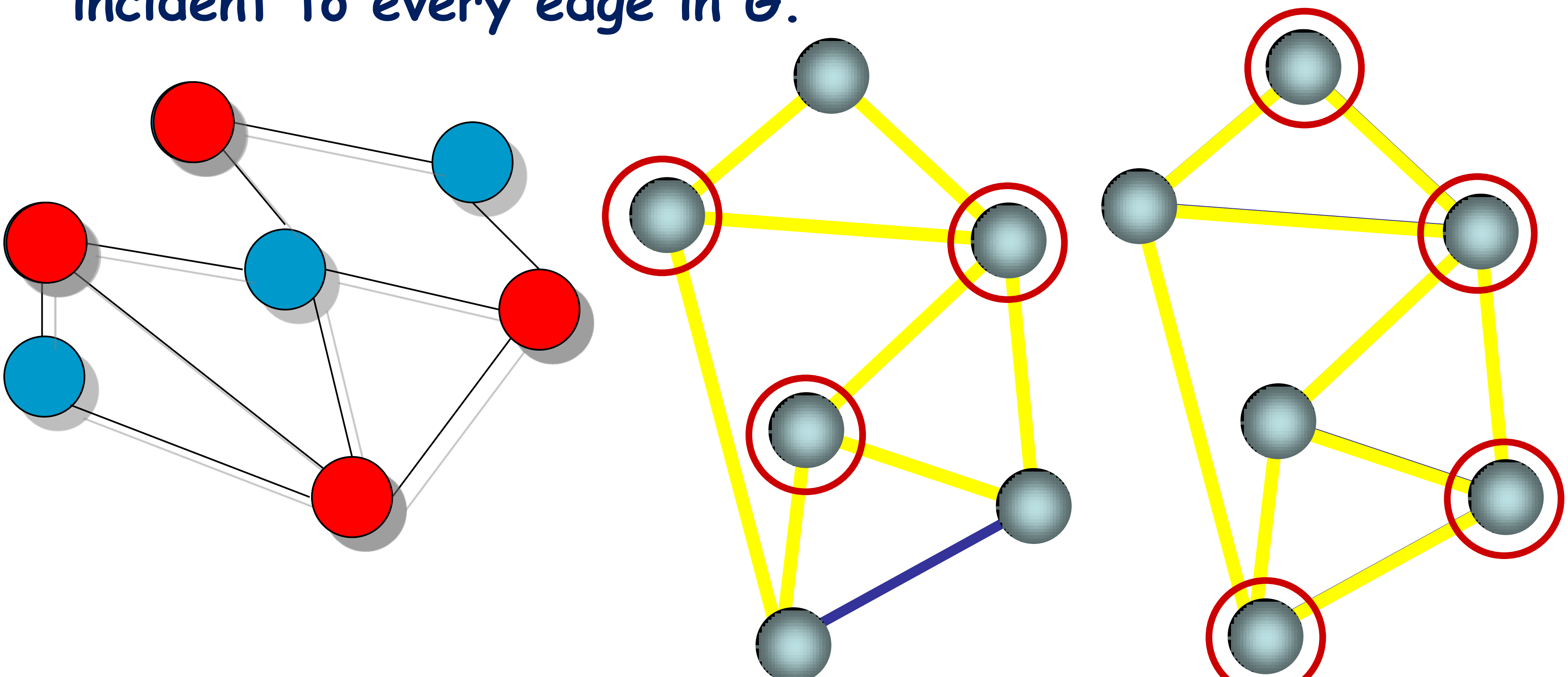
An algorithm returning a near-optimal solution in polynomial time is called approximate algorithm.



# Approximation Algorithms

## Vertex Cover

A *vertex cover* for a graph G is a subset of vertices incident to every edge in G.



Size of a vertex cover: the number of vertices in it.

# Approximation Algorithms

## Vertex Cover

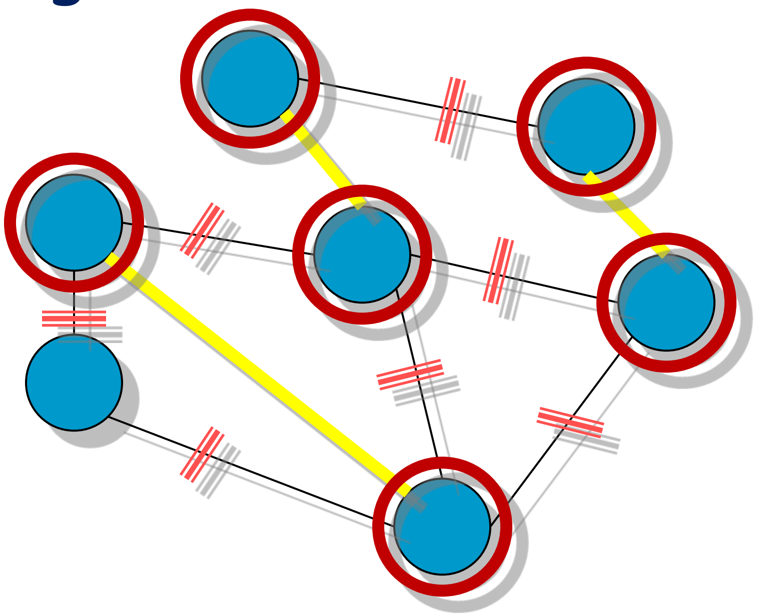
## How to find?

**Idea:**

Repeatedly pick an arbitrary edge (u, v)

Add its endpoints u and v to the vertex-cover set

Remove all edges incident on u or v



# Approximation Algorithms

## Vertex Cover Problem

### Optimization Problem

What is the minimum size vertex cover in $G$?

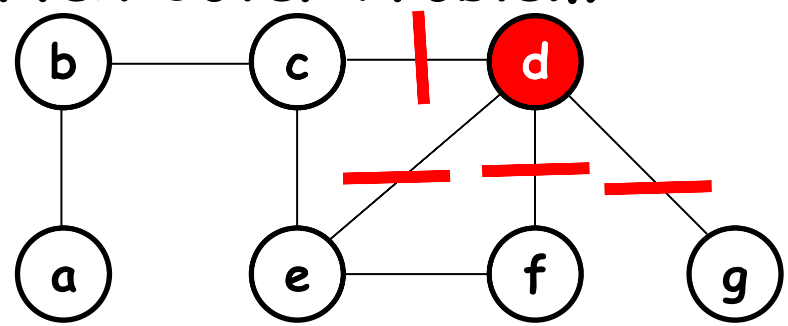
### Restated as a decision problem:

Does a vertex cover of size $k$ exist in $G$?

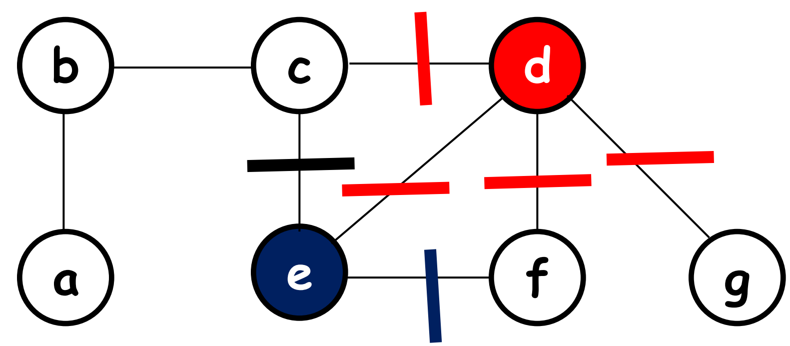
# Approximation Algorithms

## Optimal Solution of Vertex Cover Problem

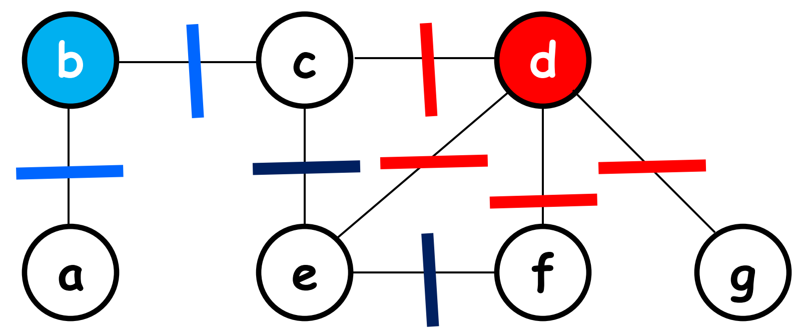
The vertex d covers edges dc,de,df,dg




The vertex e covers edges ef, and ec




The vertex b covers edges bc and ba




The optimal vertex cover for this problem contains only three vertices: *b, d, and e.*

# Approximation Algorithms

## Approx. Solution of Vertex Cover Problem

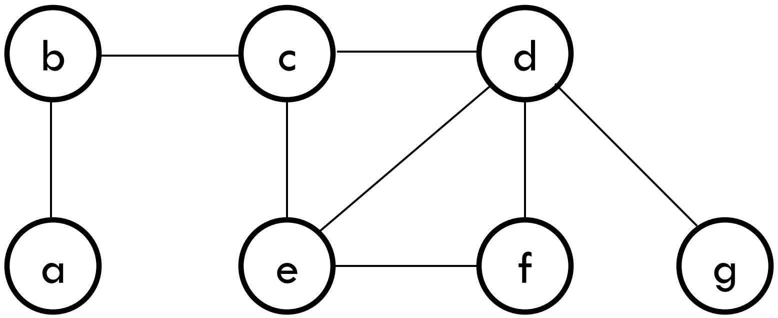
Alg.: APPROX-VERTEX-COVER(G)

1. $C \leftarrow \varnothing$
2. $E' \leftarrow E[G]$
3. **while** $E' \neq \varnothing$
4. do choose (u, v) arbitrary from E'
5. $C \leftarrow C \cup \{u, v\}$
6. remove from E' all edges incident on u, v
7. **return** $C$

# Approximation Algorithms

## Approx. Solution of Vertex Cover Problem

### Example:




1. $C \leftarrow \varnothing$

   $C = \varnothing$

2. $E' \leftarrow E[G]$

   $E' = \{ ab, bc, cd, ce, de, df, dg, ef \}$

# Approximation Algorithms

## Approx. Solution of Vertex Cover Problem

### Example:

3. **while** $E' \neq \varnothing$

4. do choose (u, v) arbitrary from E'

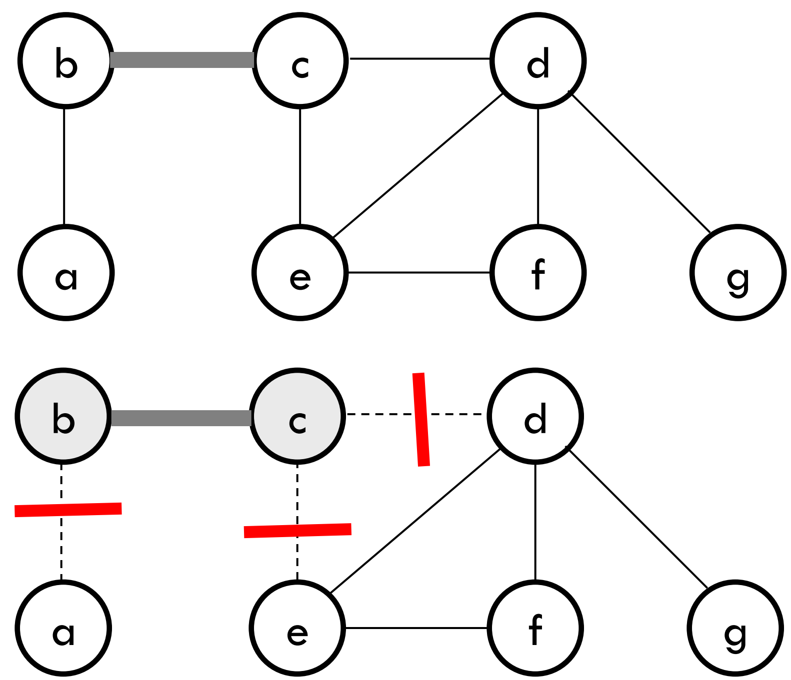
$E'$={ ab, bc, cd, ce, de, df, dg, ef }

5. $C \leftarrow C \cup \{u, v\}$

C={bc}

6. remove from E' all edges incident on u, v

E'={de,df,dg,ef}



# Approximation Algorithms

## Approx. Solution of Vertex Cover Problem

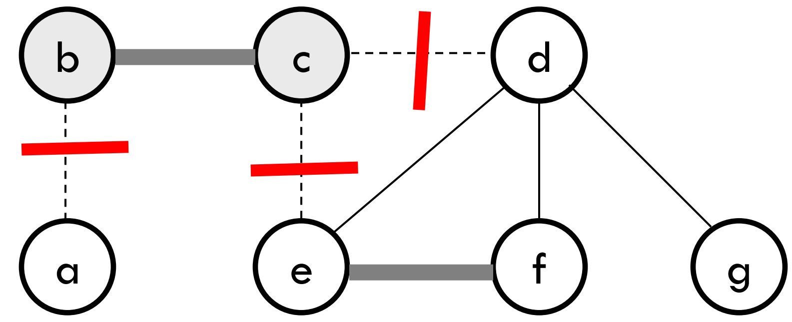
### Example:

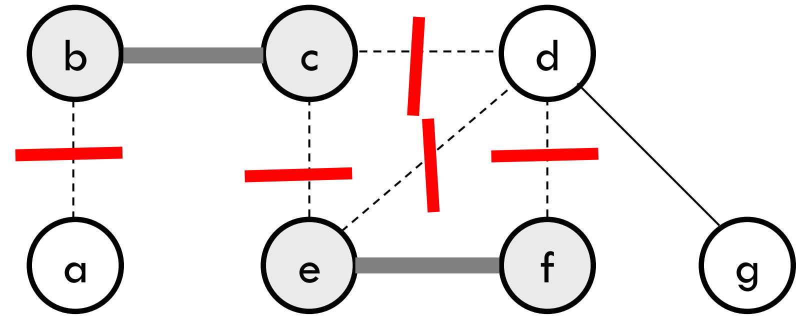
C={bc}

E'={de,df,dg,ef}

C={bc, ef}

E'={dg}

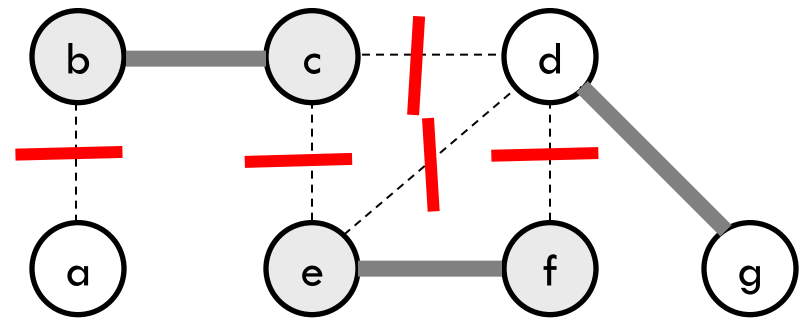


# Approximation Algorithms

## Approx. Solution of Vertex Cover Problem

### Example:




C={bc, ef}

E'={dg}

C={bc, ef, dg}

E'={}

7. **return** C

C={bc, ef, dg}

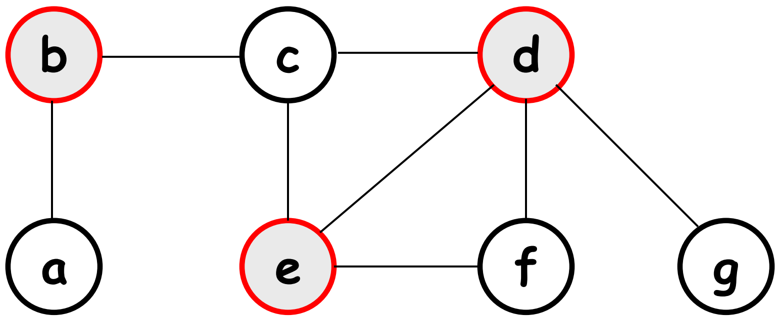
# Approximation Algorithms

## Approx. Solution of Vertex Cover Problem

### Example:

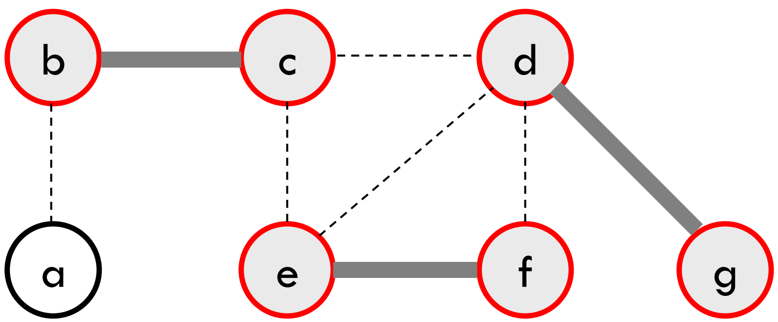
Optimal VERTEX-COVER:

The optimal vertex cover for this problem contains only three vertices:
*b, d, and e.*




APPROX-VERTEX-COVER:

*The set C, which is the vertex cover produced by* APPROX-VERTEX-COVER, contains the six vertices *b, c, d, e, f, g.*




The approximation algorithm returns an optimal solution that is no more than twice the optimal vertex cover

# The Set Covering Problem

Let $X = \{1, \ldots, 6\}$

Subsets of X: $F=\{S_1,S_2,S_3,S_4,S_5,S_6\}$

Let $S_1 = \{1,2,4\}$, $S_2 = \{3,4,5\}$, $S_3 = \{1,3,6\}$,

$S_4 = \{2,3,5\}$, $S_5 = \{4,5,6\}$, and $S_6 = \{1,3\}$

The minimum number of subsets of in F that covers all the elements of X is 3($S_1$,$S_2$, $S_3$)

We are given:

A finite set $X = \{1, \ldots, n\}$

A collection of subsets of X, F: $S_1, S_2, \ldots, S_m$

Find a minimum-size subset $C \subseteq F$ that covers all the elements in X. $C=\{S_1,S_2,S_3\}$

# The Set Covering Problem

## Restated as Decision Problem:

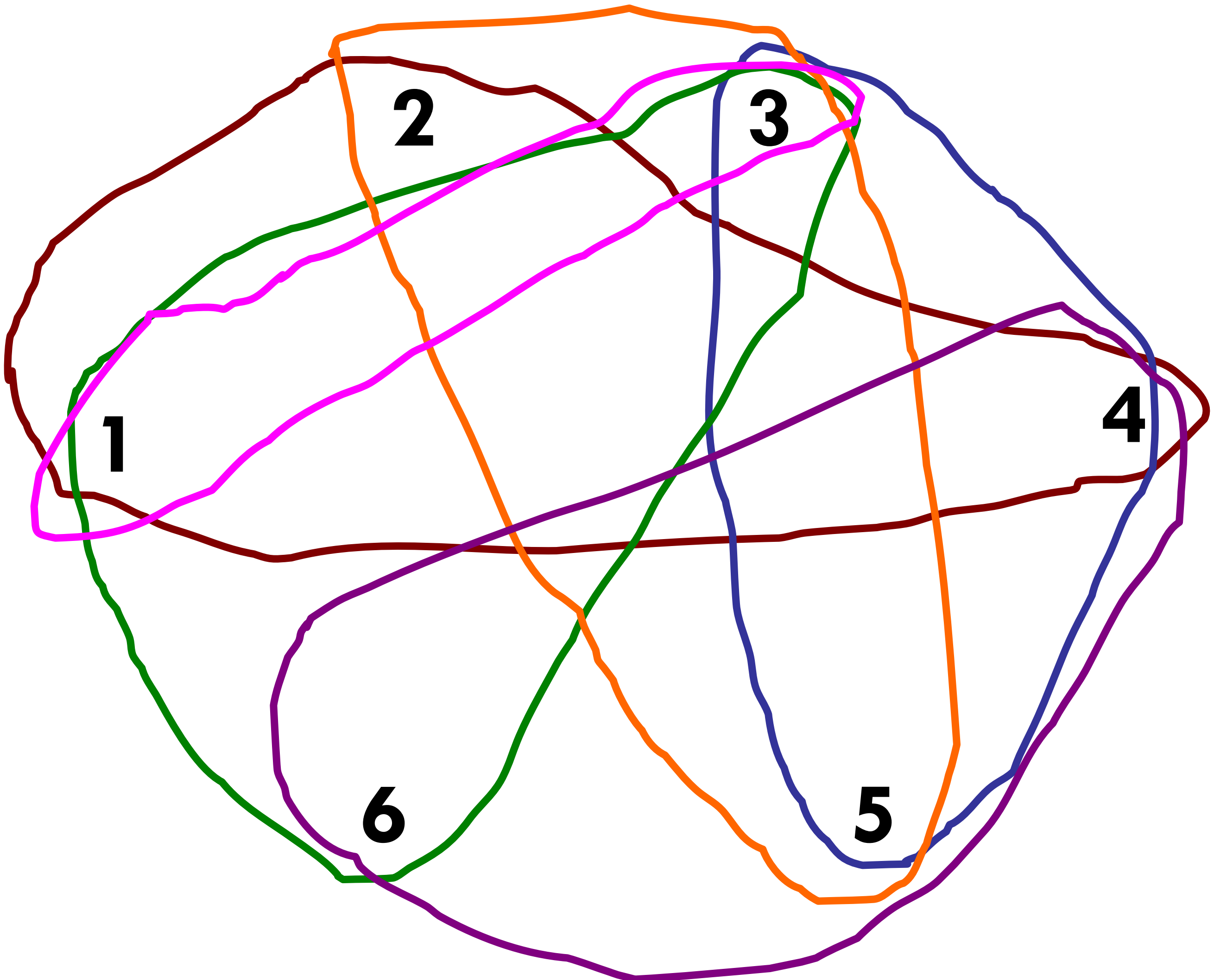
Given a number k find if there exist k sets $S_{i1}$, $S_{i2}$, …, $S_{ik}$ such that:

$$S_{i1} \cup S_{i2} \cup \dots \cup S_{ik} = X$$

Learn DAA : From B K Sharma

# The Set Covering Problem

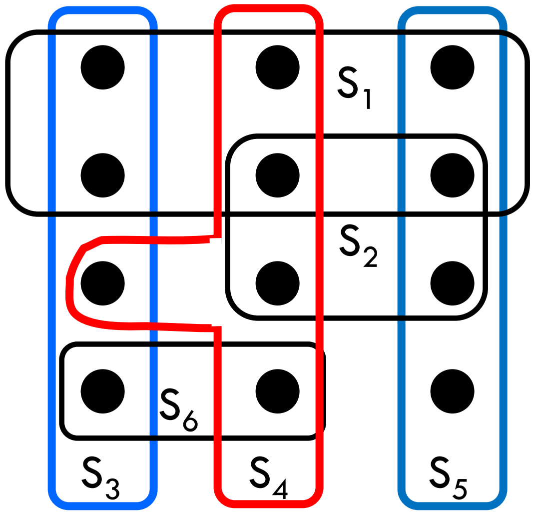
$S_1 = \{1,2,4\}$, $S_2 = \{3,4,5\}$, $S_3 = \{1,3,6\}$, $S_4 = \{2,3,5\}$, $S_5 = \{4,5,6\}$, $S_6 = \{1,3\}$



# The Set Covering Problem

**Idea:**

**At each step, pick a set S that covers the greatest number of remaining elements.**




**Optimal: $C = \{S_3, S_4, S_5\}$**

# The Set Covering Problem

## Algorithm

Alg.: GREEDY-SET-COVER(X, F)

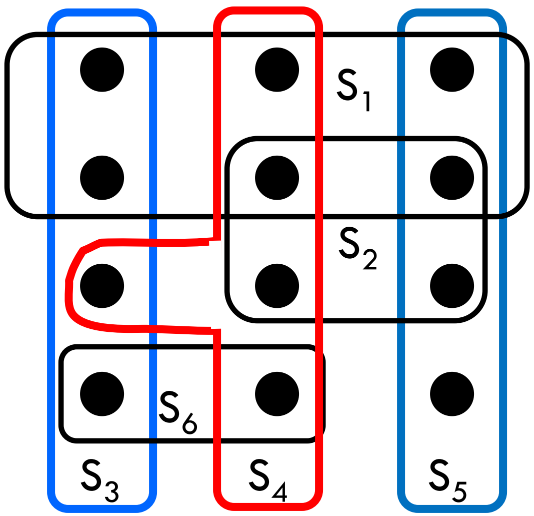
1. $U \leftarrow X$
2. $C \leftarrow \varnothing$
3. **while** $U \neq \varnothing$ **do**
4. select an $S \in F$ that maximizes $|S \cap U|$
5. $U \leftarrow U - S$
6. $C \leftarrow C \cup \{S\}$
7. **return** $C$

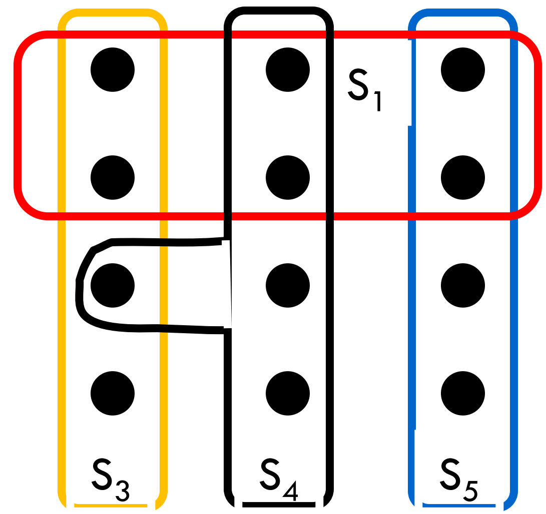
Learn DAA : From B K Sharma

# The Set Covering Problem

## Algorithm

## Example



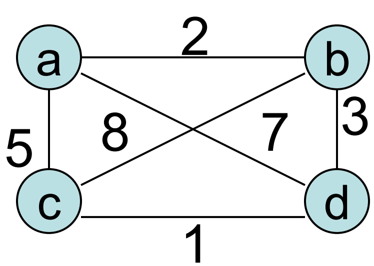

**Optimal:** $C = \{S_3, S_4, S_5\}$

**Approx.** $C=\{S_1, S_4, S_5, S_3\}$

# The Travelling Salesman Problem (TSP)

Find the shortest tour through n cities with known positive integer distances between them.

Given a graph G, find the shortest simple circuit (Hamiltonian circuit) in G which passes through all the vertices.




| Tour | Length |
|---|---|
| abcda | 18 |
| abdca | 11 |
| … | |

Totally there are P(4,4)=4! tours.

Generally, there may have *n!* tours in a graph of *n* vertices and if we check all the tours the computing time is $O(n!)$.

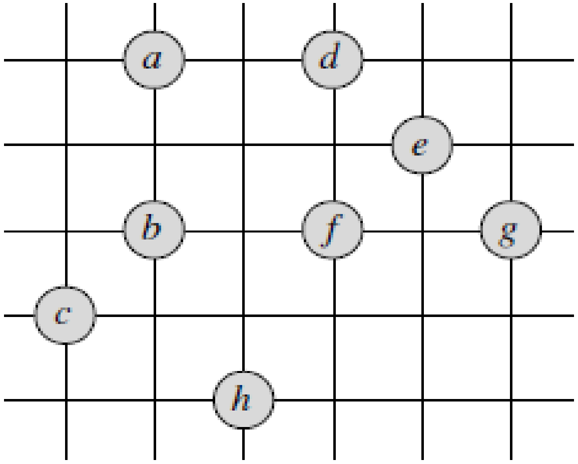
People couldn't find a good algorithm (using polynomial time) to solve it!

## Approximation Algorithm: APPROX-TSP-TOUR($G$, $c$)

1. Select a vertex $r \in V[G]$ to be a "root" vertex
2. Compute a minimum spanning tree $T$ for $G$ from root $r$ using MST-PRIM($G$, $c$, $r$)
3. Let $L$ be the list of vertices visited in a preorder tree walk of $T$
4. Return the Hamiltonian cycle $H$ that visits the vertices in the order $L$

# Approximation Algorithm: APPROX-TSP-TOUR(*G*, *c*)

## Example



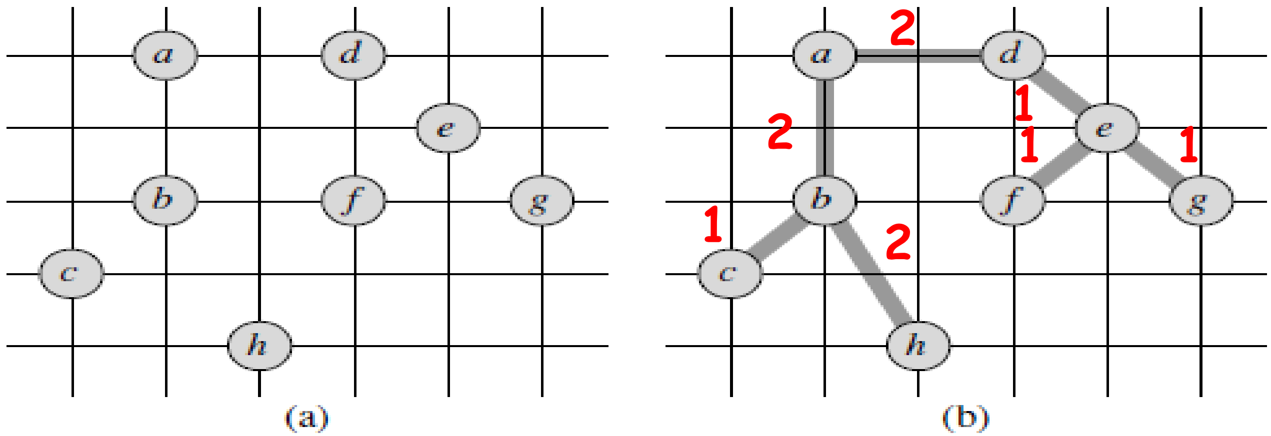

(a)

Part (a) of the figure shows the given set of vertices.

The ordinary euclidean distance is used as the cost function between two points.

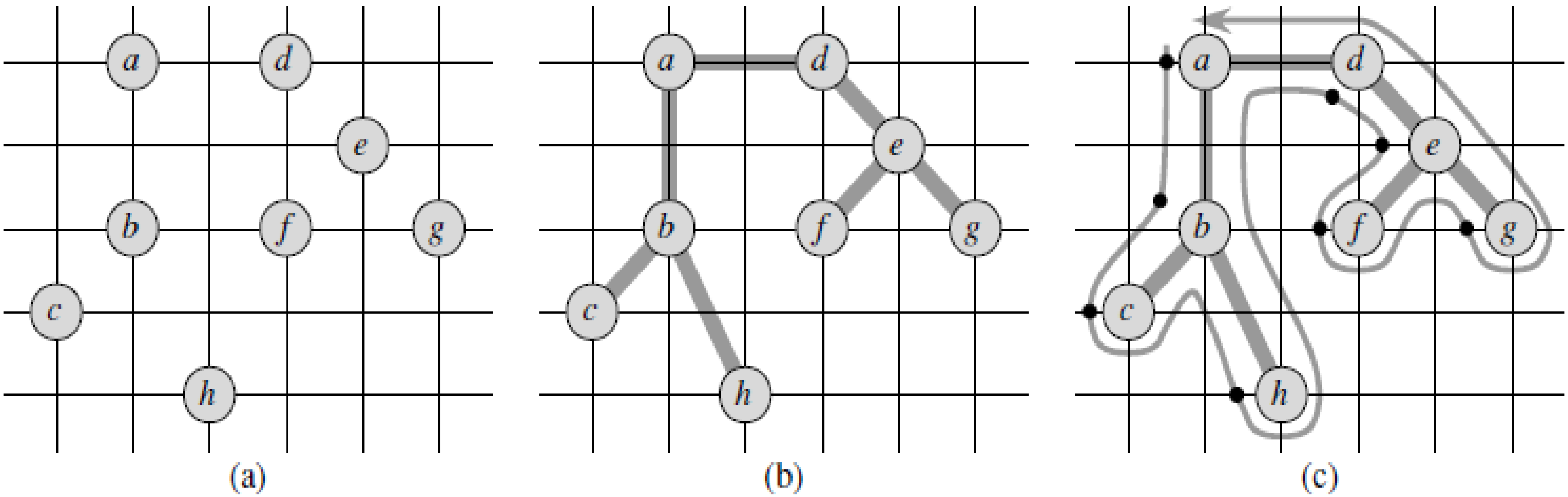
For example, *f is one unit to the right and two units up from h.*

# Approximation Algorithm: APPROX-TSP-TOUR(G, c)




**Part (b) shows the minimum spanning tree *T grown from root vertex 'a' by MST-PRIM.***

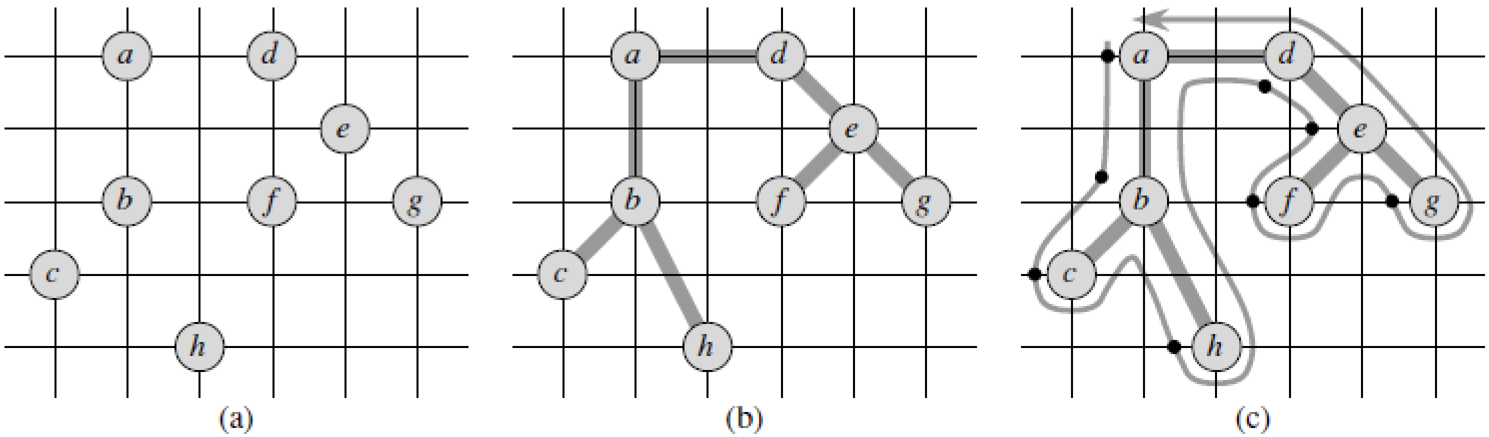
# Approximation Algorithm: APPROX-TSP-TOUR(*G*, *c*)

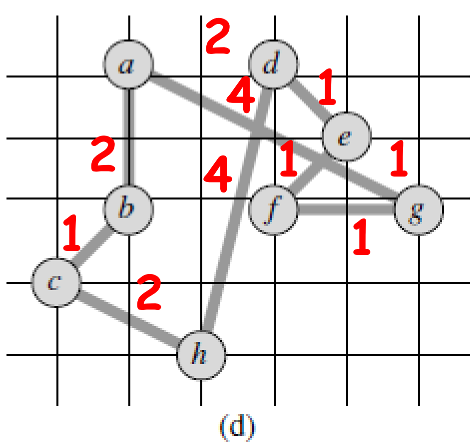



Part (c) shows how the vertices are visited by a preorder walk of *T*, *starting at 'a'*.

*A full walk of the tree visits the vertices in the order a, b, c, b, h, b, a, d, e, f, e, g, e, d, a.*

# Approximation Algorithm: APPROX-TSP-TOUR(G, c)



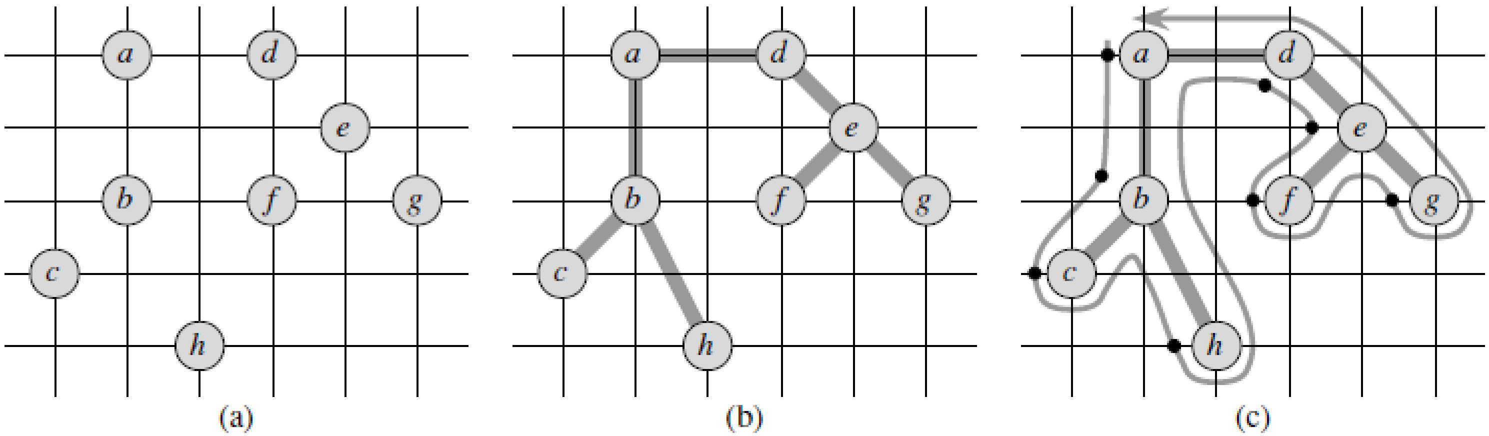



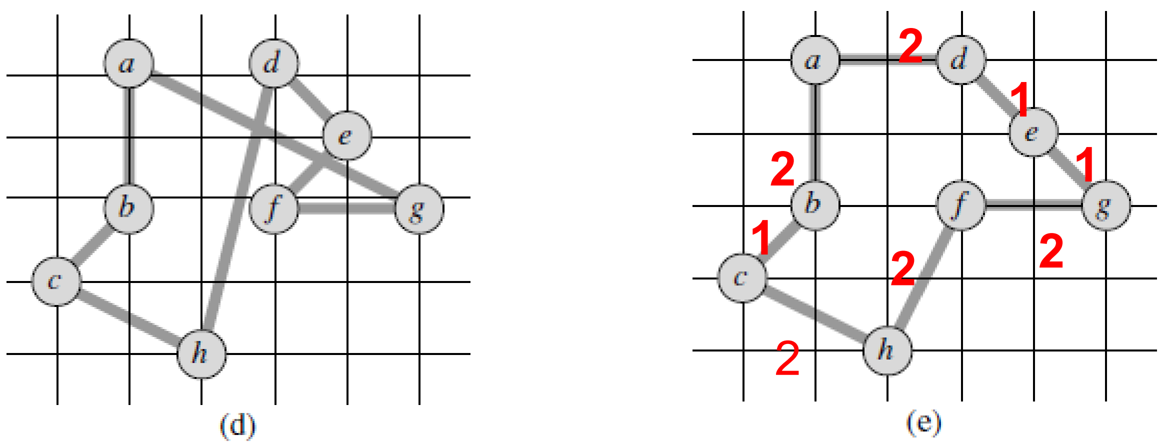

A tour of the vertices obtained by visiting the vertices in the order given by the preorder walk.

a, b, c, h, d, e, f, g, a

Part (d) displays the corresponding tour, which is the tour returned by APPROX-TSP-TOUR. Its total cost is approximately 19.074.

# Approximation Algorithm: APPROX-TSP-TOUR(G, c)




The running time of APPROX-TSP-TOUR is $O(V^2)$.

Part (e) displays an optimal tour for the given set of vertices. Its total cost is approximately 13.715., which is about 23% shorter.