Computer Algorithms

Unit-5

NP Hard and NP Complete Problems

Basic Concepts

- Problems that can be solved by a polynomial time algorithm
- Sorting – O(n log n)
- Searching – O(log n)
- Problems for which no polynomial time algorithm is known
- Traveling salesperson $O(2^n n^2)$
- No one has been able to develop a polynomial time algorithm for such problems.

Deterministic and Non-deterministic Algorithms

- Deterministic Algorithms:-
- Result of every operation is uniquely defined
- E.g. Normal Quick Sort

- Non-deterministic Algorithms:-
- Algorithm contains some operations whose outcomes are not uniquely defined
- But limited to specified sets of possibilities
- It is allowed to choose any one of these outcomes
- E.g. Randomized Quick Sort

Definitions

- Decision Problem –
- Any problem for which the answer is either 0 or 1

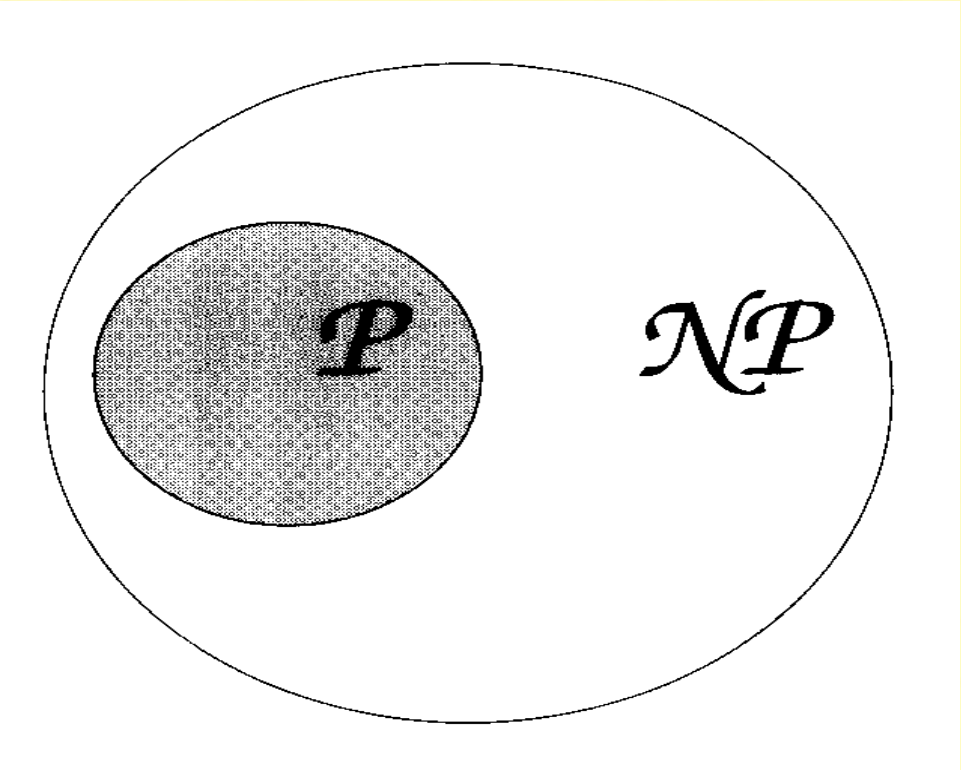
- Optimization Problem –
- Any problem that involves the identification of an optimal value of a given cost function

- Decision Algorithms
- Optimization Algorithms

Definitions

- P is the set of all the problems solvable by deterministic algorithms in polynomial time.
- NP is the set of all the problems solvable by nondeterministic algorithms in polynomial time.

- Since deterministic algorithms are just a special case of nondeterministic ones
- conclude that P is subset of NP
- Optimization problems are very complex.
- Mostly, it is not possible to solve Optimization problems in polynomial time.


P
NP

Reducibility

- Let L1 and L2 be problems.
- Problem L1 reduces to L2 L1 α L2
- if and only if there is a way to solve L1 by a deterministic polynomial time algorithm
- using a deterministic algorithm that solves L2 in polynomial time.
- This definition implies that
- if we have a polynomial time algorithm for L2,
- then we can solve L1 in polynomial time.
- Reducibility is transitive (if $L_1 \propto L_2$ and $L_2 \propto L_3$, then $L_1 \propto L_3$)

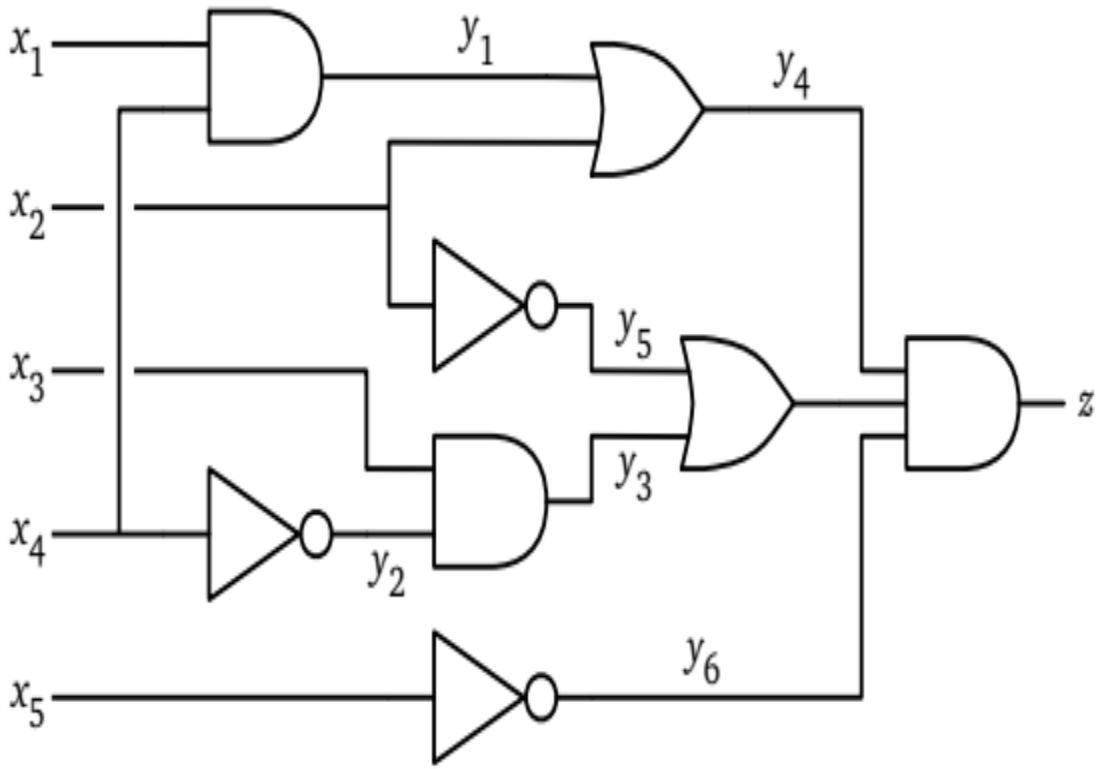
- To prove that problem P is NP-Hard –
- Reduce that problem to known NP-Hard Problem.
- Satisfiability (SAT) Problem is NP-Hard.

Satisfiability

- Let x_1, x_2, x_3,… denote Boolean variables
- their value is either true or false
- Let $\overline{x_i}$ denote the negation of x_i.
- A literal is either a variable or its negation.
- A formula in the propositional calculus is an expression that can be constructed using literals and the operations and / or.
- Examples of such formulas are

$$(x_1 \wedge \bar{x}_2) \vee (x_3 \wedge \bar{x}_4) \text{ and } (x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2)$$

- The symbol V denotes or and Λ denotes and.



$$(y_1 = x_1 \wedge x_4) \wedge (y_2 = \overline{x_4}) \wedge (y_3 = x_3 \wedge y_2) \wedge (y_4 = y_1 \vee x_2) \wedge$$
$$(y_5 = \overline{x_2}) \wedge (y_6 = \overline{x_5}) \wedge (y_7 = y_3 \vee y_5) \wedge (z = y_4 \wedge y_7 \wedge y_6) \wedge z$$

0/1 knapsack problem using n=3, m=6

(w1, w2, w3)=(2, 3, 4) and (p1, p2, p3)=(1, 2, 5)

I1	I2	I3	
T	T	T	
T	T	F	
T	F	T	T
T	F	F	
F	T	T	
F	T	F	
F	F	T	
F	F	F	

Satisfiability

- Propositional Calculus – Mathematical Logic
- Satisfiability problem is
- to determine whether a formula is true for some assignment of truth values to the variables.

Satisfiability

- It is easy to obtain a polynomial time nondeterministic algorithm that terminates successfully
- if and only if a given propositional formula E(x1,...,xn) is satisfiable.
- Such an algorithm could proceed by simply choosing (non-deterministically) one of the 2^n possible assignments of truth values to (x1,..., xn)
- and verifying that E(x1,..., xn) is true for that assignment.

Basic Concepts

- Non-Polynomial – time needed is exponential
- Requires vast amount of time
- Even moderate-size problems can not be solved
- Do not provide a method to obtain polynomial time algorithms for problems
- many of the problems for which there are no known polynomial time algorithms are computationally related
- Two classes of problems
- NP – Hard NP – Complete

- A problem that is NP-complete has the property –
- It can be solved in polynomial time
- if and only if all other NP-complete problems can also be solved in polynomial time.

- If an NP-hard problem can be solved in polynomial time
- then all NP-complete problems can be solved in polynomial time.
- All NP-complete problems are NP-hard,
- But some NP-hard problems are not known to be NP-complete

NP-Complete NP-Hard

- A problem L is NP-hard
- if and only if satisfiability reduces to L

- A problem L is NP-hard
- if and only if it can be reduced to satisfiability problem (SAT)

- As Satisfiability Problem (SAT) is NP- Hard, Problem L is also NP-Hard

NP-Complete NP-Hard

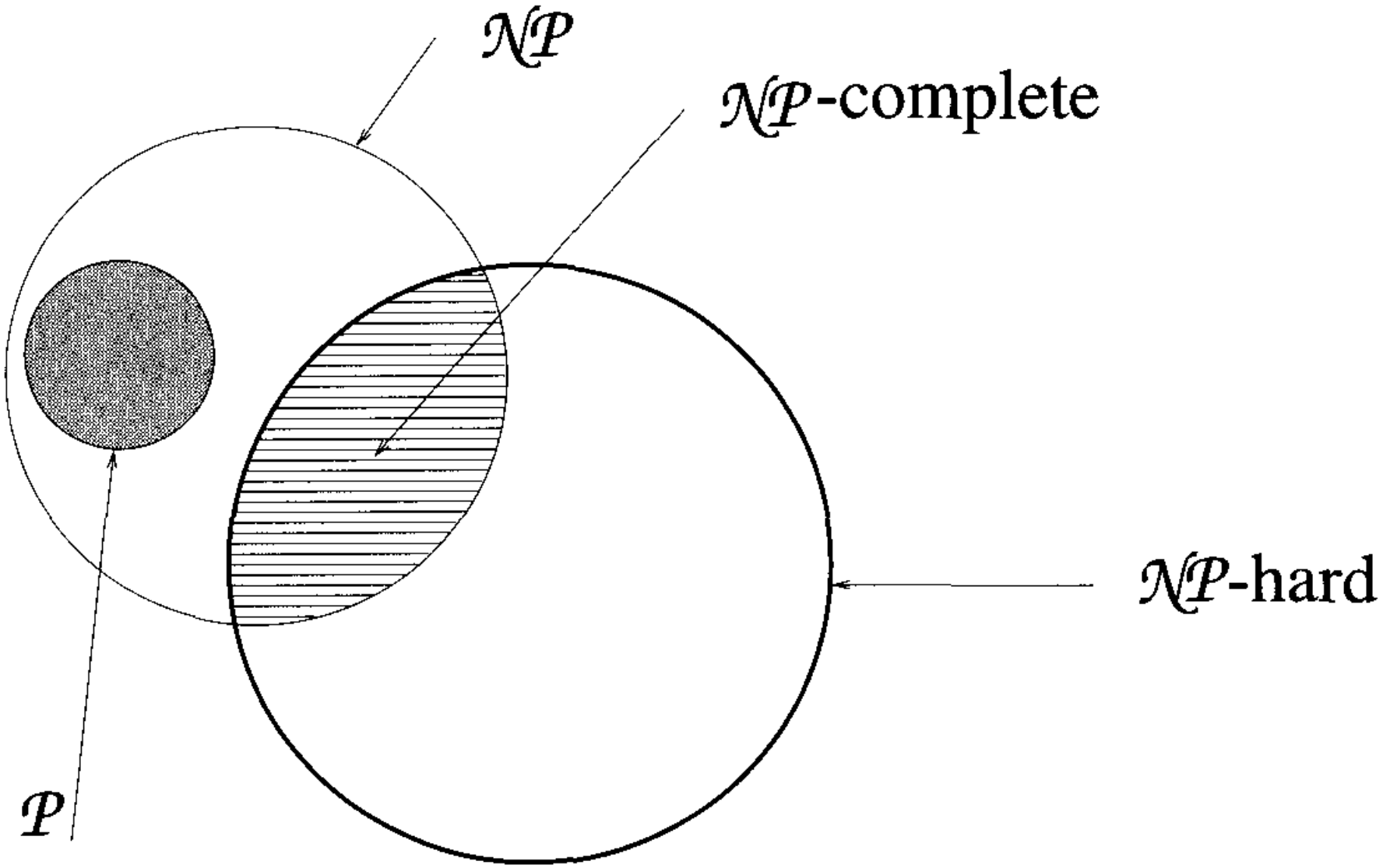
- Set of very hard problems
- If any NP Hard problem can be solved in polynomial time, then all NP-Complete problems can be solved in polynomial time.
- It means, it is impossible to solve NP-Hard problem in polynomial time.

NP-Complete NP-Hard

- A problem L is NP-complete if and only if
- L is NP-hard and L belongs to NP

- NP-Complete problems are less hard than NP-Hard problems.
- Smaller instance of NP-Hard problem can form NP-Complete Problem.
- Up to certain extent, it can be solved in polynomial time.
- E.g. Decision Problem instance of Optimization problem.

- Only Decision problems can be NP-complete. – 0 or 1
- an optimization problem may be NP-hard.
- if L1 is a decision problem and L2 an optimization problem,
- L1 is reducible to L2
- E.g. knapsack decision problem reduces to the knapsack optimization problem.



- Commonly believed NP-complete, and NP-hard problems relationship among P, NP,

Cook's Theorem

- Satisfiability is in P if and only if P = NP
- From Satisfiability definition – satisfiability is in NP
- If P=NP then satisfiability is in P

- We have to show how to obtain from any polynomial time nondeterministic algorithm A and input I
- a formula Q(A, I) such that Q is satisfiable
- iff A has a successful termination with input I.

Cook's Theorem

- Deterministic algorithm Z to determine the outcome of A on any input I can be obtained.
- Algorithm Z computes Q and
- then determine whether Q is satisfiable.
- If Q obtained by deterministic algorithm Z is satisfiable then P=NP
- As in both types of algorithms deterministic and non-deterministic for that problem satisfiability reduce to it
- Means – Matching truth table values at both sides
- It can be shown that if satisfiability is in P, then P =NP.

NP Hard Graph Problems

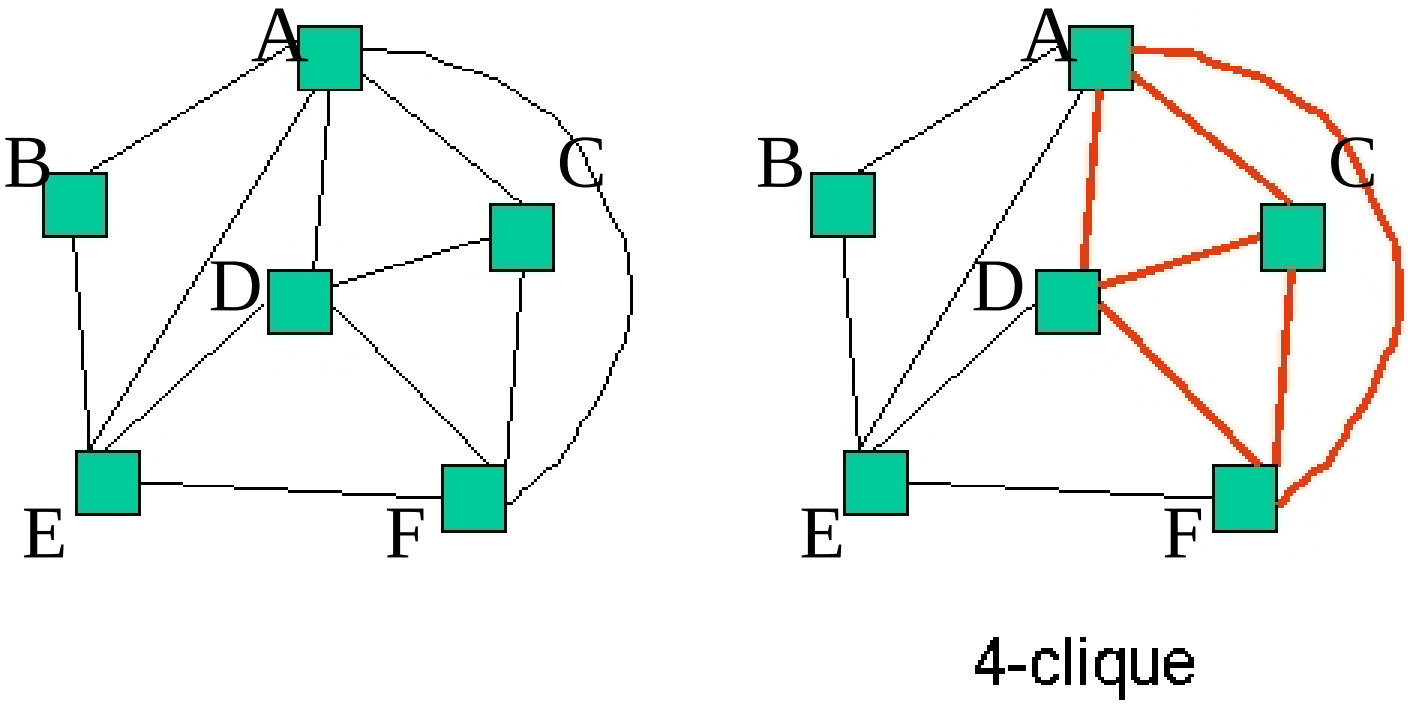
- Strategy is:
- 1. Pick a problem L2 already known to be NP-hard.
- 2. Show how to obtain an instance I' of L2 from any instance I of L1
- such that from the solution of I' we can determine the solution to instance I of L1
- 3. Conclude from step (2) that L1 is reducible to L2.
- 4. Conclude from steps (1) and (3) and the transitivity of reducibility, that L1 is NP-hard.

Clique Decision Problem (CDP)

- A maximum complete subgraph of a graph G=(V,E) is a clique –
- Sets of elements where each pair of elements is connected.
- The size of clique is the number of vertices in it
- Max clique problem is an optimization problem that to determine the size of a largest clique in G
- Decision problem is to determine whether G has a clique of size at least k for some given k

Clique Decision Problem (CDP)

Clique Example



4-clique

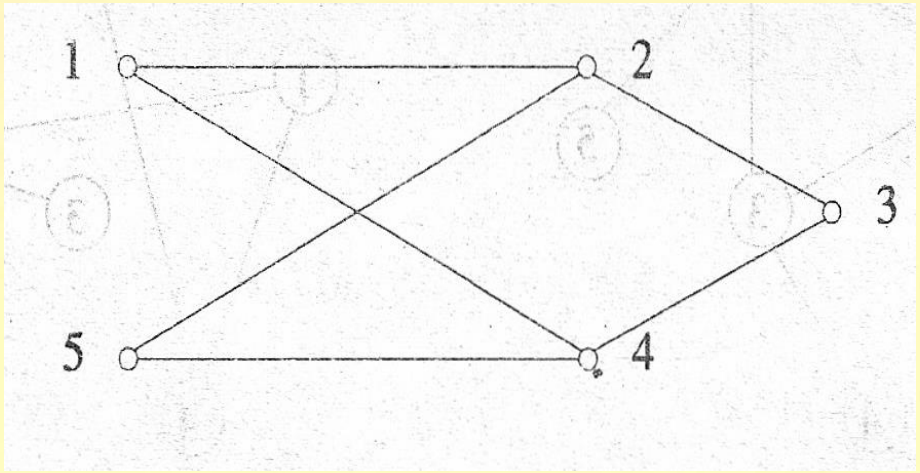
Clique Decision Problem (CDP)

- Max clique problem can be solved in polynomial time
- if and only if the clique optimization problem can be solved in polynomial time

- Satisfiability reduces to Clique Problem
- Clique Problem is NP-Hard
- Since CDP ϵ NP, CDP is also NP-Complete

Node Cover Decision Problem (NCDP)

- A set S subset of V
- is a node cover for a graph G = (V, E)
- if and only if all edges in E are incident to at least one vertex in S.
- The size |S| of the cover is the number of vertices in S.
- Example –
- Consider the graph of Figure
- S = {2,4} is a node cover of size 2.
- S = {1,3,5} is a node cover of size 3.



- In the node cover decision problem
- given a graph G and an integer k.
- required to determine whether G has a node cover of size at most k

- Theorem - The clique decision problem is reducible to the node cover decision problem.

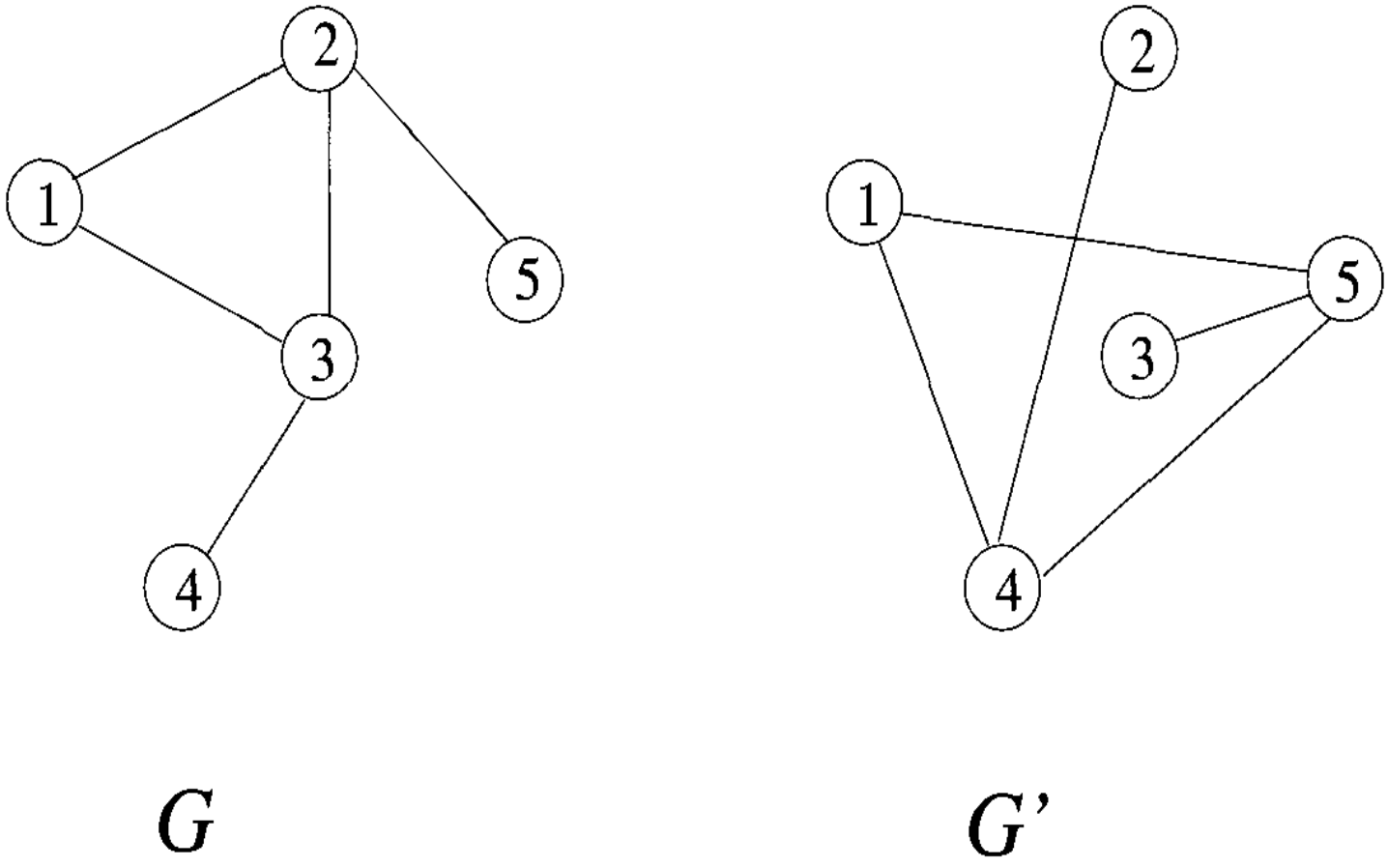


Figure 11.6 A graph and its complement

Proof

- Let G = (V, E) and k define an instance of CDP.
- Assume that |V| = n.
- We construct a graph G’ such that G’ has a node cover of size at most n - k if and only if G has a clique of size at least k.
- Graph G' is given
- by G' = (V, $\overline{E}$), where $\overline{E}$ = {{u,v) | u ϵ V,v ϵ V and (u,v) $\notin$ E}.
- The set G' is known as the complement of G.

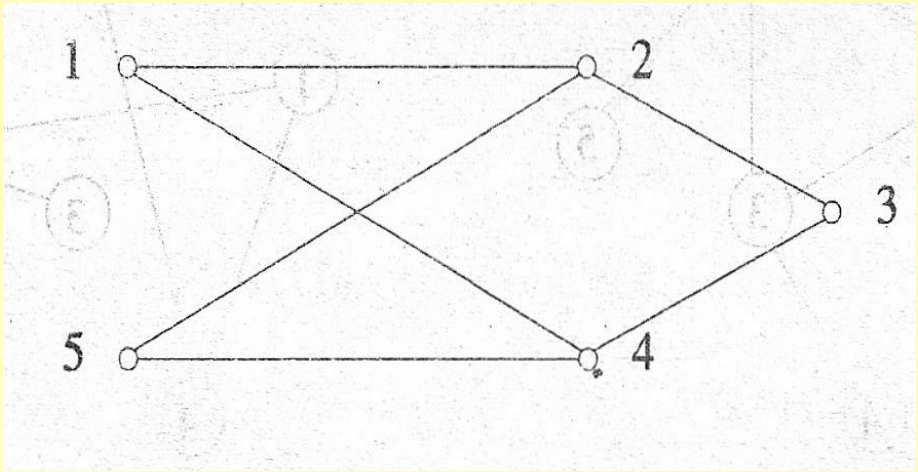
- G’ has a node cover {4, 5} of size 2
- since every edge of G’ is incident either on the node 4 or on the node 5.
- G has a clique of size 5-2 = 3
- consisting of the nodes 1,2, and 3.

Chromatic Number Decision Problem (CNDP)

- A coloring of a graph G = (V, E)
- is a function f: V $\rightarrow$ {1, 2,... , k}
- defined for all i ϵ V.
- If (u, v) ϵ E, then f(u) $\neq$ f(v).
- The chromatic number decision problem is to determine whether G has a coloring for a given k.

- It is NP-Hard

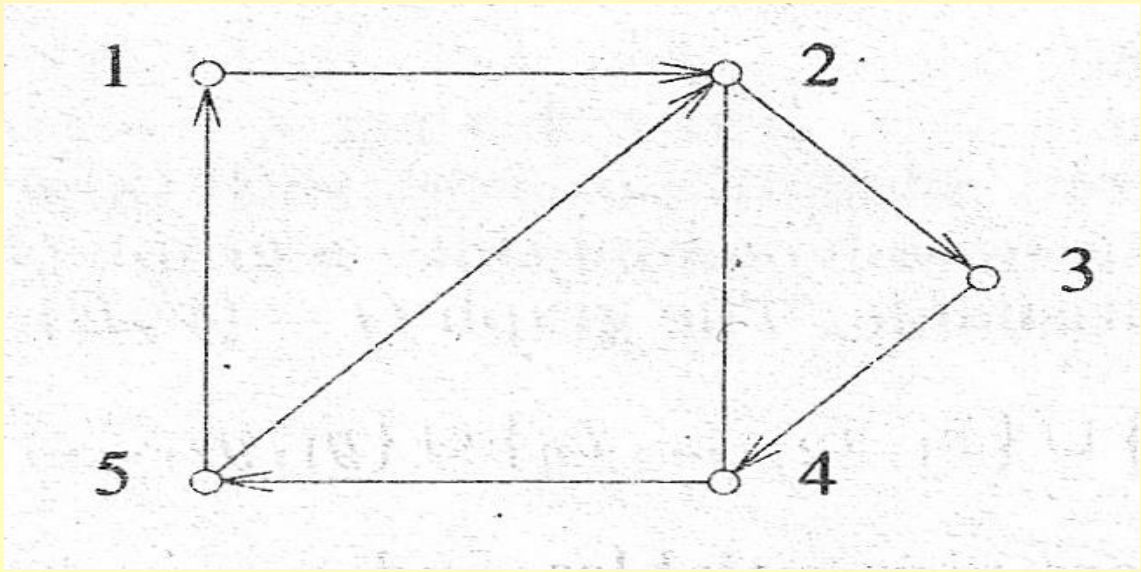
- A possible 2-coloring of the graph of Figure is
- f(1) = f(3) = f(5) =1.
- and f(2) = f(4) = 2.
- this graph has no 1-coloring possible.



Directed Hamiltonian Cycle (DHC)

- A directed Hamiltonian cycle in a directed graph
- G = (V, E)
- is a directed cycle of length m = |V|
- the cycle goes through every vertex exactly once and then returns to the starting vertex.
- The DHC problem is to determine whether G has a directed Hamiltonian cycle

- Satisfiability reduces to directed Hamiltonian Cycle
- So it is NP-Hard problem



- 1, 2, 3, 4, 5, 1 is a directed Hamiltonian cycle in the graph
- If the edge (5, 1) is deleted from this graph, then it has no directed Hamiltonian cycle.

Traveling Salesperson Decision Problem (TSP)

- Traveling salesperson decision problem is
- to determine whether a complete directed graph G=(V, E)
- with edge costs c(u, v) has a tour of cost at most M.

- Theorem –
- Directed Hamiltonian cycle (DHC) is reducible to the traveling salesperson decision problem (TSP)

Traveling Salesperson Decision Problem (TSP)

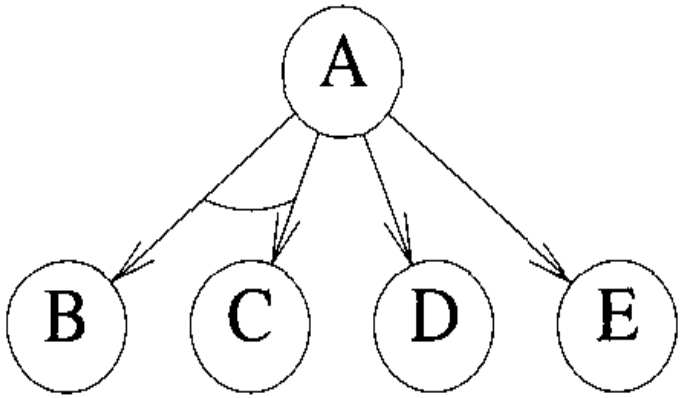
- Theorem –
- Directed Hamiltonian cycle (DHC) is reducible to the traveling salesperson decision problem (TSP)

Proof: From the directed graph $G = (V, E)$ construct the complete directed graph $G' = (V, E')$, $E' = \{\langle i, j\rangle \mid i \neq j\}$ and $c(i, j) = 1$ if $\langle i, j\rangle \in E$; $c(i, j) = 2$ if $i \neq j$ and $\langle i, j\rangle \notin E$. Clearly, G' has a tour of cost at most n iff G has a directed Hamiltonian cycle. □

AND/OR Graph Decision Problem (AOG)

- complex problems can be broken down into a series of sub problems such that
- the solution of all or some of these results in the solution of the original problem.
- These sub problems can be broken down further into sub-sub-problems, and so on,
- until the only problems remaining are sufficiently primitive as to be trivially solvable.

- This breaking down of a complex problem into several subproblems can be represented by a directed graphlike structure
- in which nodes represent problems and descendants of nodes represent the subproblems associated with them



(a)

problem can be solved by solving either both the sub-problems B and C
or the single subproblem D or E.

AND/OR Graph Decision Problem (AOG)

- The AND/OR graph decision problem (AOG) is
- to determine whether G has a solution graph of cost at most k,
- for k a given input.

NP Hard Scheduling Problems

- SUM OF SUBSETS PROBLEM
- Suppose we are given *n* distinct positive numbers
- we desire to find all combinations of these numbers whose sums are m.
- This is called the sum of subsets problem.
- NP-Hard problem

Example 7.6 Figure 7.10 shows the portion of the state space tree generated by function SumOfSub while working on the instance $n = 6$, $m = 30$, and $w[1 : 6] = \{5, 10, 12, 13, 15, 18\}$. The rectangular nodes list the values of s, k, and r on each of the calls to SumOfSub. Circular nodes represent points at which subsets with sums m are printed out. At nodes A, B, and C the output is respectively (1, 1, 0, 0, 1), (1, 0, 1, 1), and (0, 0, 1, 0, 0, 1). Note that the tree of Figure 7.10 contains only 23 rectangular nodes. The full state space tree for $n = 6$ contains $2^6 - 1 = 63$ nodes from which calls could be made (this count excludes the 64 leaf nodes as no call need be made from a leaf). □

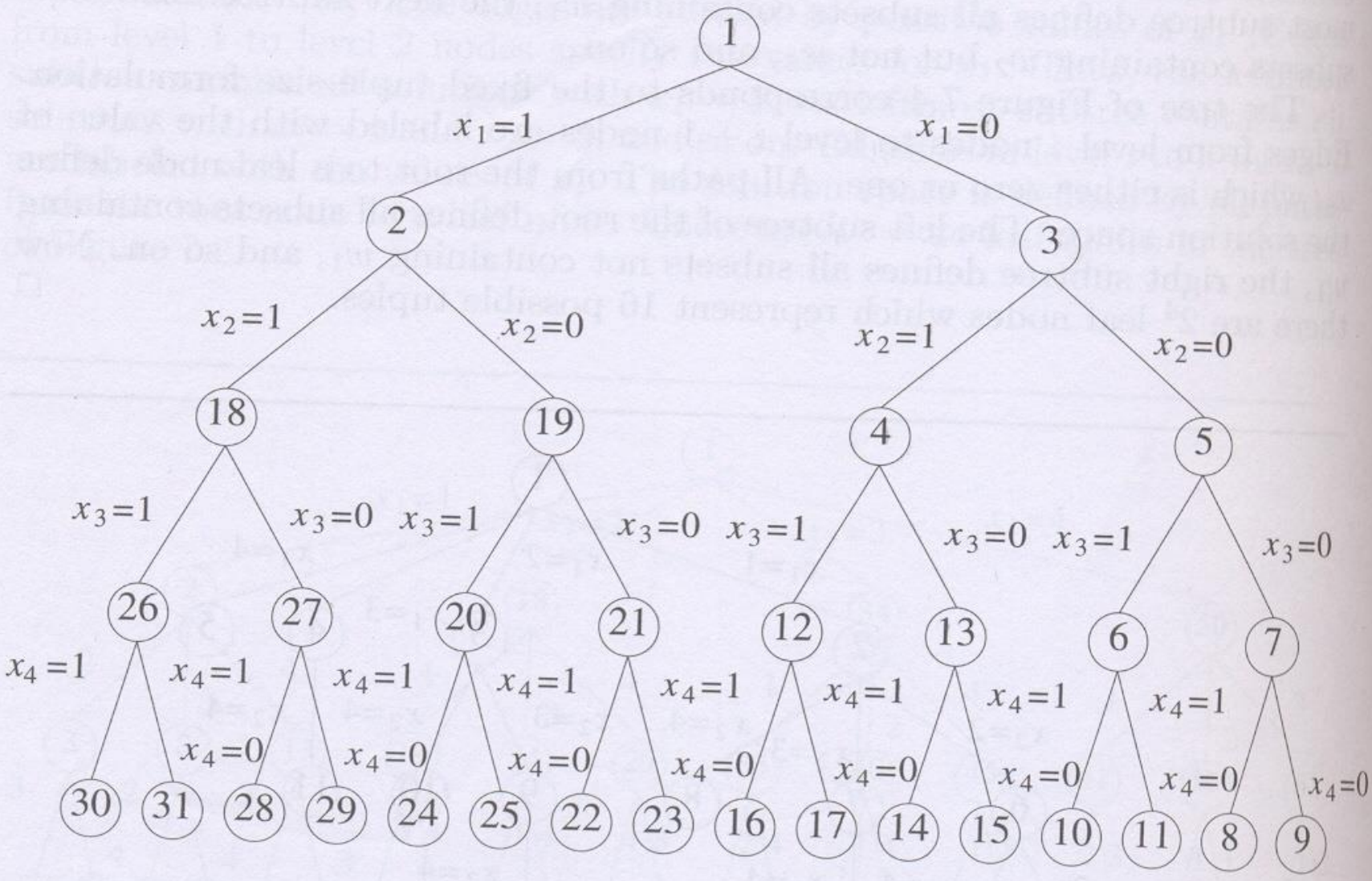


Figure 7.4 Another possible organization for the sum of subsets problems. Nodes are numbered as in *D*-search.

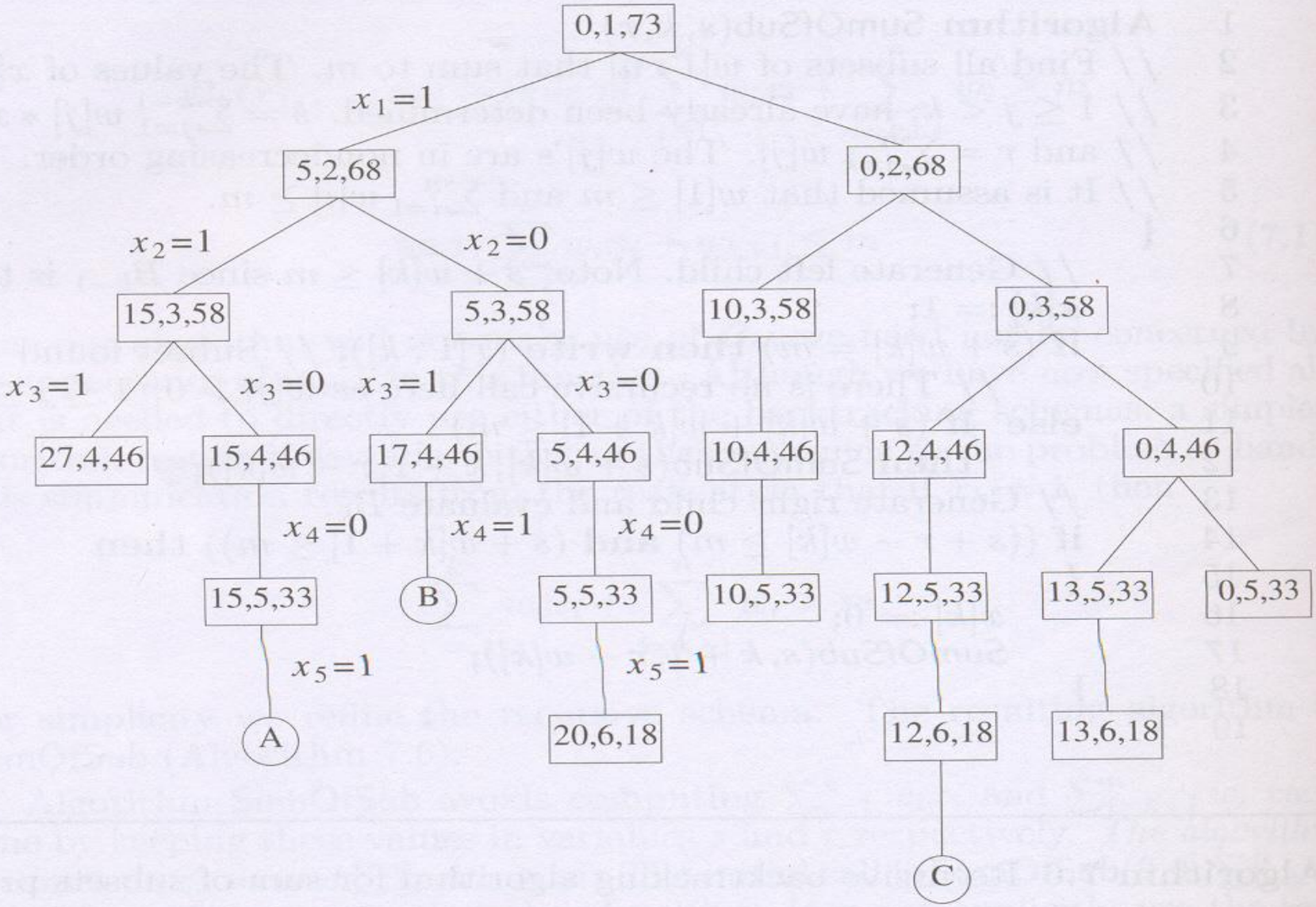


Figure 7.10 Portion of state space tree generated by SumOfSub

Partition Problem

- To decide whether a given set
- $A = \{a_1, a_2, \ldots, a_n\}$
- of n positive integers
- has a partition P such that
- $\sum_{i \in P} a_i = \sum_{i \in \overline{P}} a_i$
- Reducible to sum of subsets problem
- So, Partition Problem is NP-Hard problem

Scheduling Identical Processors

- Set of n jobs – J_i $1 \leq i \leq n$
- each job i has processing time t_i
- set of m identical machines- P_k $1 \leq k \leq m$
- Schedule S is an assignment of jobs to processors
- For each job Ji, S specifies the time intervals and the processors on which this job is to be processed.
- A job cannot be processed by more than one processor at any given time.

- Let f_i be the time at which the processing of job Ji is completed.
- The mean finish time (MFT) of schedule is

$$\mathrm{MFT}(S) = \frac{1}{n} \sum_{1 \le i \le n} f_i$$

- Let Ti be the time at which Pi finishes processing all jobs assigned to it.
- The finish time (FT) of S is

$$\mathrm{FT}(S) = \max_{1 \le i \le m} \{T_i\}$$

- Schedule S is a non-preemptive schedule
- if and only if each job Ji is processed continuously from start to finish on the same processor.
- In a preemptive schedule each job need not be processed continuously to completion on one processor.

- Find minimum Finish Time Schedule.

- Theorem:
- Partition α minimum finish time non-preemptive schedule.
- Proof:
- We prove this for m = 2.
- Let a_i, $1 \leq i \leq n$, be an instance of the partition problem.
- Define n jobs with processing requirements $t_i = a_i$
- There is a non-preemptive schedule for this set of jobs on two processors with finish time at most $\Sigma\ t_i /2$
- iff there is a partition of the a_i s

- Example
- Consider the following input to the partition problem:
- $a_1 = 2$, $a_2 = 5$, $a_3 = 6$, $a_4 = 7$, and $a_5 = 10$.
- The corresponding minimum finish time non-preemptive schedule problem has the input
- $t_1 = 2$, $t_2 = 5$, $t_3 = 6$, $t_4 = 7$, and $t_5 = 10$.
- There is a non-preemptive schedule for this set of jobs with finish time 15:

- P1 takes the jobs J_2 and J_5;
- P2 takes the jobs J_1, J_3, and J_4.
- This solution yields a solution for the partition problem also:
- $\{a_2,a_5\},\{a_1,a_3,a_4\}$.

- When m = 2, minimum finish time schedules can be obtained in 0(n log n) time if n jobs are to be scheduled.
- When m = 3, obtaining minimum finish time schedules (whether preemptive or non-preemptive) is a NP-hard.

Flow Shop Scheduling

Problem Statement

- n jobs requiring m tasks
  $T_{1i}, T_{2i}, T_{3i}, \ldots T_{mi}$, $1 \leq i \leq n$
- Task T_{ji} is to be performed on Processor P_j, $1 \leq i \leq m$
- Time required to complete T_{ji} is t_{ji}
- Schedule for n jobs is an assignment of task to time interval on the processor.

Problem statement: Conditions

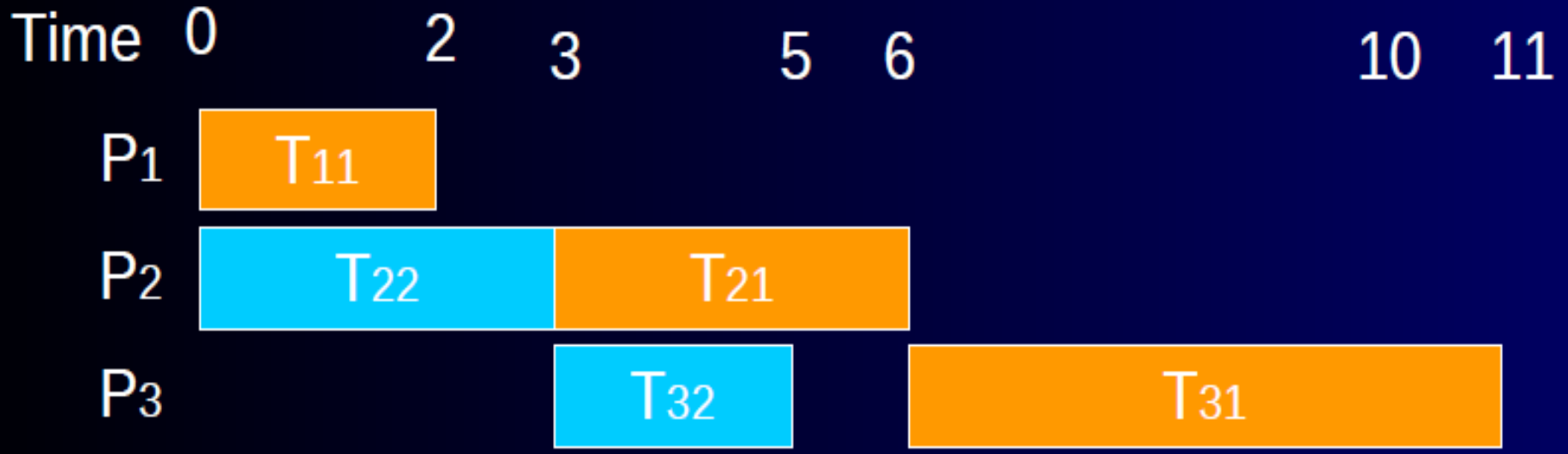
- Task T_{ji} must be asigned to the Processor P_j
- No processor may have more than one task assigned to it in any time interval.
- For any job i the Processing of task T_{ji}, $j \geq 1$, cannot be started until task $T_{j-1, i}$ is completed.

Example

- Two jobs have to be scheduled on three processors.
- Task time is given by matrix

	Job1	Job 2
Processor 1	2	0
Processor 2	3	3
Processor 3	5	2

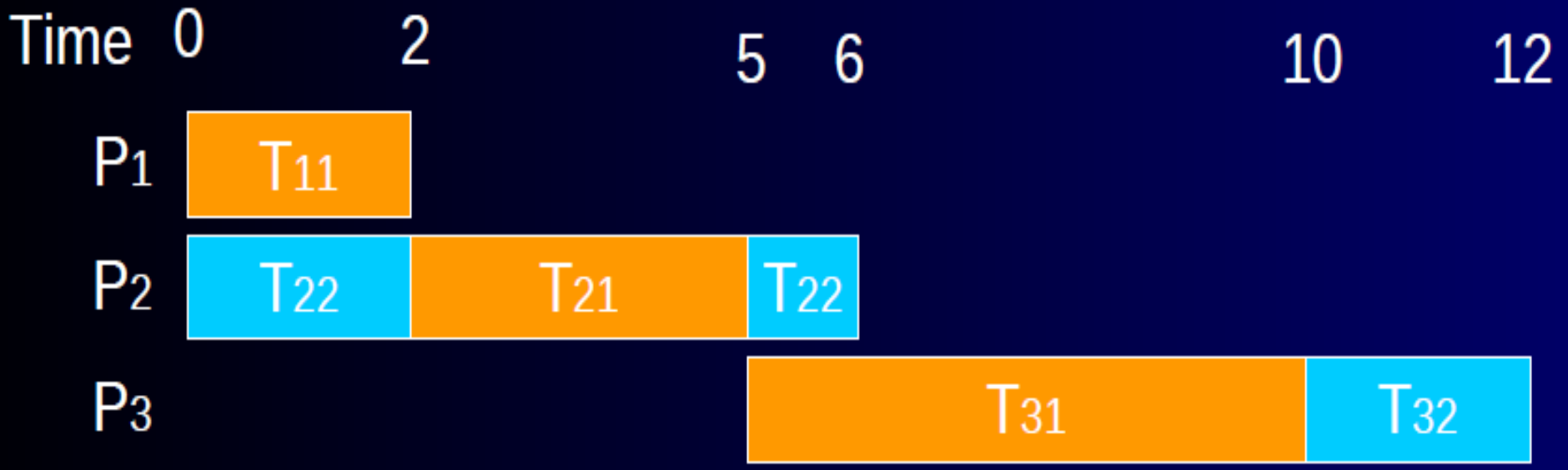
Schedule 1 (Non Preemptive)



	J1	J2
P1	2	
0		
P2	3	
3		
P3	5	
2		

Schedule 2(Preemptive)

	J1	J2
P1	2	
	0	
P2	3	
	3	
P3	5	
	2	



Schedule

- Non-preemptive schedule is a schedule in which processing of task on any processor is not terminated until it is complete. (Schedule 1)
- A schedule in which it is not true is preemptive schedule. (Schedule 2)

Flow Shop Scheduling

Example 5.28 Let $n = 4, (a_1, a_2, a_3, a_4) = (3, 4, 8, 10)$, and $(b_1, b_2, b_3, b_4) = (6, 2, 9, 15)$. The sorted sequence of a's and b's is $(b_2, a_1, a_2, b_1, a_3, b_3, a_4, b_4) = (2, 3, 4, 6, 8, 9, 10, 15)$. Let $\sigma_1, \sigma_2, \sigma_3$, and σ_4 be the optimal schedule. Since the smallest number is b_2, we set $\sigma_4 = 2$. The next number is a_1 and we set $\sigma_1 = a_1$. The next smallest number is a_2. Job 2 has already been scheduled. The next number is b_1. Job 1 has already been scheduled. The next is a_3 and we set σ_3. This leaves σ_3 free and job 4 unscheduled. Thus, $\sigma_3 = 4$. □

Job Shop Scheduling

- Job shop, as like a flow shop, has m different processors.
- The n jobs to be scheduled require the completion of several tasks.
- The time of the j^{th} task for job Ji is $t_{k,i,j}$
- Task j is to be performed on processor P_k.
- The tasks for any job J_i are to be carried out in the order 1, 2, 3,..., and so on.
- Task j cannot begin until task j - 1 (if j > 1) has been completed.

- it is quite possible for a job to have many tasks that are to be performed on the same processor.
- In a non-preemptive schedule, a task once begun is processed without interruption until it is completed.
- Obtaining either a minimum finish time preemptive schedule or a minimum finish time non-preemptive schedule is a NP-Hard even when m = 2.

NP-Hard Code Generation Problems

- The function of a compiler is to translate programs written in some source language into an equivalent machine language program.
- Thus, the C++ compiler on the Sparc 10 translates C++ programs into the machine language of this machine.
- Consider the translation of arithmetic expressions in a language such as C++ into assembly language code.

- The translation clearly depends on the particular assembly language (and hence machine) being used.
- To begin, we assume a very simple machine model.
- We call this model machine A.
- This machine has only one register called the accumulator.
- All arithmetic has to be performed in this register.
- Θ represents a binary operator such as +, - ,*, and /
- then the left operand of Θ must be in the accumulator.

- For simplicity, we restrict ourselves to these four operators.
- The discussion easily generalizes to other operators.
- The relevant assembly language instructions are:
- LOAD X
- load accumulator with contents of memory location X.
- STORE X
- store contents of accumulator into memory location X.
- OP X
- OP may be ADD, SUB, MPY, or DIV.

- The instruction OP X computes the operator OP using the contents of the accumulator as the left operand
- and that of memory location X as the right operand.
- As an example, consider the arithmetic expression (a+b)/(c+d).
- Two possible assembly language versions of this expression are given in Figure.
- Tl and T2 are temporary storage areas in memory.

LOAD	a	LOAD	c
ADD	b	ADD	d
STORE	$T1$	STORE	$T1$
LOAD	c	LOAD	a
ADD	d	ADD	b
STORE	$T2$	DIV	$T1$
LOAD	$T1$		
DIV	$T2$		
(a)		(b)	

Two possible codes for $(a + b)/(c + d)$

- In both the cases the result is left in the accumulator.

- Code (a) is two instructions longer than code (b).
- If each instruction takes the same amount of time, then code (b) will take 25% less time than code (a).
- For the expressions (a + b)/(c + d) and the given machine A,
- it is easy to see that code (b) is optimal.

- A translation of an expression E into the machine or assembly language of a given machine is optimal if and only if it has a minimum number of instructions.

- Definition
- A binary operator Θ is commutative in the domain D iff a Θ b = b Θ a for all a and b in D.

- Machine A can be generalized to another machine B.
- Machine B has $1 \leq$ N registers in which arithmetic can be performed.
- There are four types of machine instructions for B:
- 1. LOAD M, R
- 2. STORE M, R
- 3. OP R1, M, R2
- 4. OP R1, R2, R3

- These four instruction types perform the following functions:

1. LOAD M, R

   places the contents of memory location M into register R, $1 \leq R \leq N$.

2. STORE M, R

   stores the contents of register R, $1 \leq R \leq N$, into memory location M.

3. OP Rl, M, R2

computes contents of (R1) OP contents of (M) and places

the result in register R2.

Rl and R2 are registers;

M is a memory location.

4. OP Rl, R2, R3

computes contents of (R1) OP contents of (R2) and places the result in register R3.

Here R1, R2, and R3 are registers.

- In comparing the two machine models A and B,
- we note that when N = 1, instructions of types (1), (2) and (3) for model B are the same as the corresponding instructions for model A.
- Instructions of type (4) only allow trivial operations like a + a, a - a, a*a, and a/a to be performed without an additional memory access.
- This does not change the number of instructions in the optimal codes for A and B when N = 1.
- Hence, model A is in a sense identical to model B when N = 1.

- For model B, the optimal code for a given expression E may be different for different values of N.
- Figure shows the optimal code for the expression (a + b)/(c*d).
- Two cases are considered, N = 1 and N = 2.
- Note that when N = 1, one store has to be made whereas when N = 2, no stores are needed.
- The registers are labeled Rl and R2.
- Tl is a temporary storage location in memory.

LOAD	$c, R1$	LOAD	$c, R1$
MPY	$R1,\ d, R1$	MPY	$R1, d, R1$
STORE	$R1, T1$	LOAD	$a, R2$
LOAD	$a, R1$	ADD	$R2, b, R2$
ADD	$R1, b, R1$	DIV	$R2, R1, R1$
DIV	$R1, T1, R1$		
(a) $N = 1$		(b) $N = 2$	

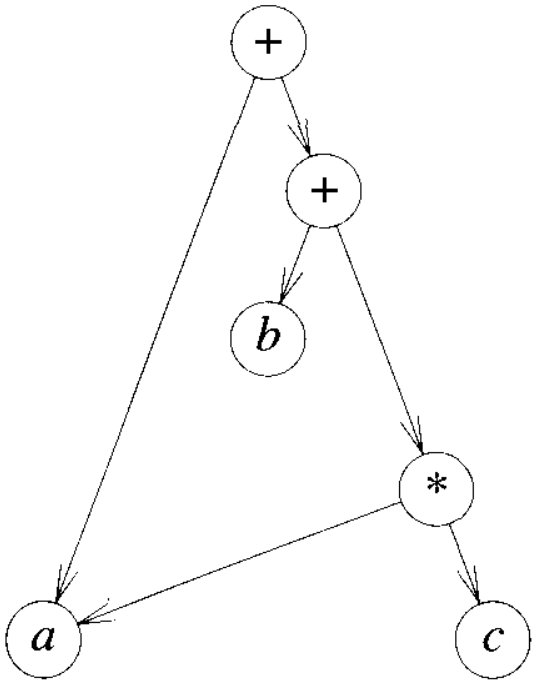
) Optimal codes for $N = 1$ and $N = 2$

- Given an expression E,
- the first question we ask is: can E be evaluated without any STOREs?
- A closely related question is:
- what is the minimum number of registers needed to evaluate E without any stores?
- These problems are NP-Hard.

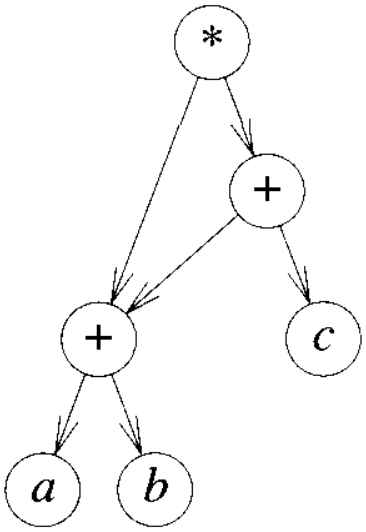
Code Generation with Common Sub-expressions

- When arithmetic expressions have common sub-expressions,
- they can be represented by a Directed Acyclic Graph (DAG).
- Every internal node (node with nonzero out-degree) in the DAG represents an operator.
- Assuming the expression contains only binary operators, each internal node P has out-degree two.

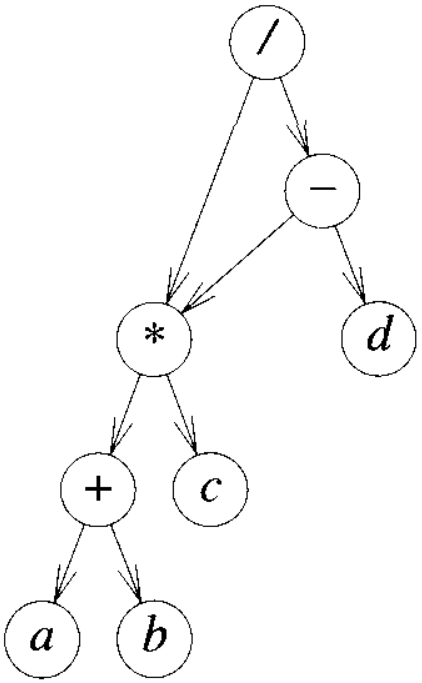
- The two nodes descendent from P are called the left and right children of P respectively.
- The children of P are the roots of the DAGs for the left and right operands of P.
- Node P is the parent of its children.
- Figure shows some expressions and their DAG representations.



$a+(b+a*c)$



$(a+b)*(a+b+c)$



$(a+b)*c/((a+b)*c-d)$

- Definition
- A Leaf is a node with out-degree zero.
- A Level-One Node is a node both of whose children are leaves.
- A Shared Node is a node with more than one parent.
- A Leaf DAG is a DAG in which all shared nodes are leaves.
- A Level-One DAG is a DAG in which all shared nodes are level-one nodes.

- Example
- The dag of Figure (a) is a Leaf DAG.
- Figure (b) is a Level-One DAG.
- Figure (c) is neither a Leaf DAG nor a Level-One DAG

- A Leaf DAG results from an arithmetic expression in which the only common sub-expressions are simple variables or constants.
- A Level-One DAG results from an expression in which the only common sub-expressions are of the form
- a Θ b, where a and b are simple variables or constants and Θ is an operator.

- The problem of generating optimal code for Level-One DAG is NP-Hard even when the machine for which code is being generated has only one register
- Determining the minimum number of registers needed to evaluate a DAG with no STORE is also NP-Hard.

- Example -The optimal codes for the DAG of Figure (b) for one and two-register machines is given
- The minimum number of registers needed to evaluate this DAG without any STORE is two.

```
LOAD    a,R1
ADD     R1,b,R1
STORE   T1,R1
ADD     R1,c,R1
STORE   T2,R1
LOAD    T1,R1
MUL     R1,T2, R1
```

(a)

```
LOAD    a,R1
ADD     R1,b,R1
ADD     R1,c,R2
MUL     R1,R2,R1
```

(b)

Optimal codes for one- and two-register machines

Implementing Parallel Assignment Instructions

- A parallel assignment instruction has the format
- $(v_1, v_2, \ldots, v_n) := (e_1, e_2, ..., e_n)$
- where the v_i's are distinct variable names and
- the e_i's are expressions.
- The semantics of this statement is that, the value of v_i is updated to be the value of the expression e_i, $1 \leq i \leq n$
- The value of the expression e_i is to be computed using the values the variables in e_i have before this instruction is executed.

- Example
- (A, B) := (B, C); is equivalent to A := B; B := C;.
- (A, B) := {B, A); is equivalent to T := A; A := B;

  B := T;.
- (A, B) := {A + B, A- B); is equivalent to Tl := A; T2 := B; A := Tl+T2; B:=Tl-T2;
- And also to Tl := A; A := A+B; B :=T1-B;

- (A, B) := {B, A) without using temporary variable
- A = A + B
- B = A – B A = A - B

- it may be necessary to store some of the v_i's in temporary locations when executing a parallel assignment.
- These stores are needed only when some of the v_i's appear in the expressions e_j's, $1 \leq j \leq n$
- A variable v_i is referenced by expression e_i if and only if v_i appears in e_j.
- only referenced variables need to be copied into temporary locations.
- Finding number of temporary locations needed is NP-Hard Problem.

Thank You

Email:- suhas.bhagate@gmail.com