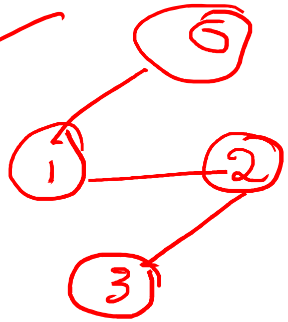
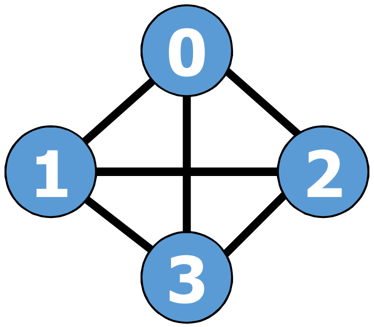
Spanning Tree (ST)

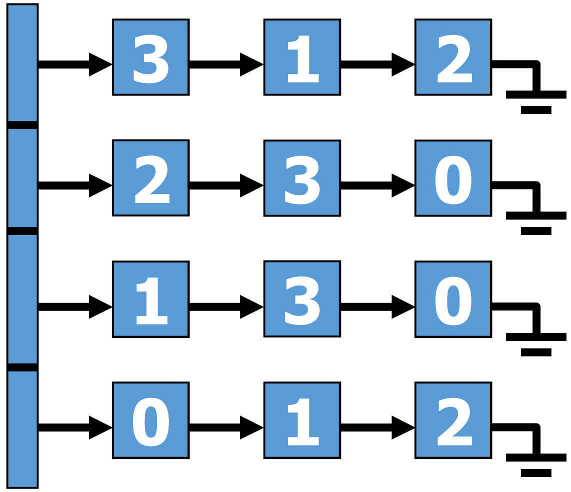
- A spanning tree is a minimal subgraph G', such that V(G')=V(G) and G' is connected.
- Either DFS or BFS can be used to create a spanning tree

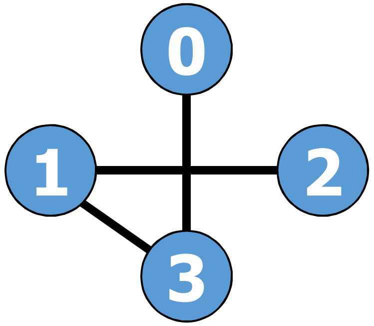
$V(G1) = \{0, 1, 2, 3\}$



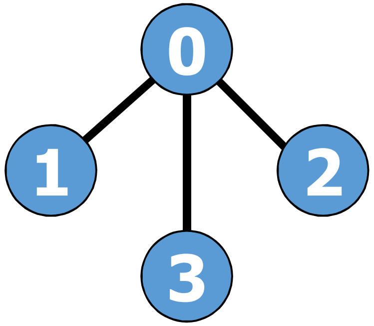


G1



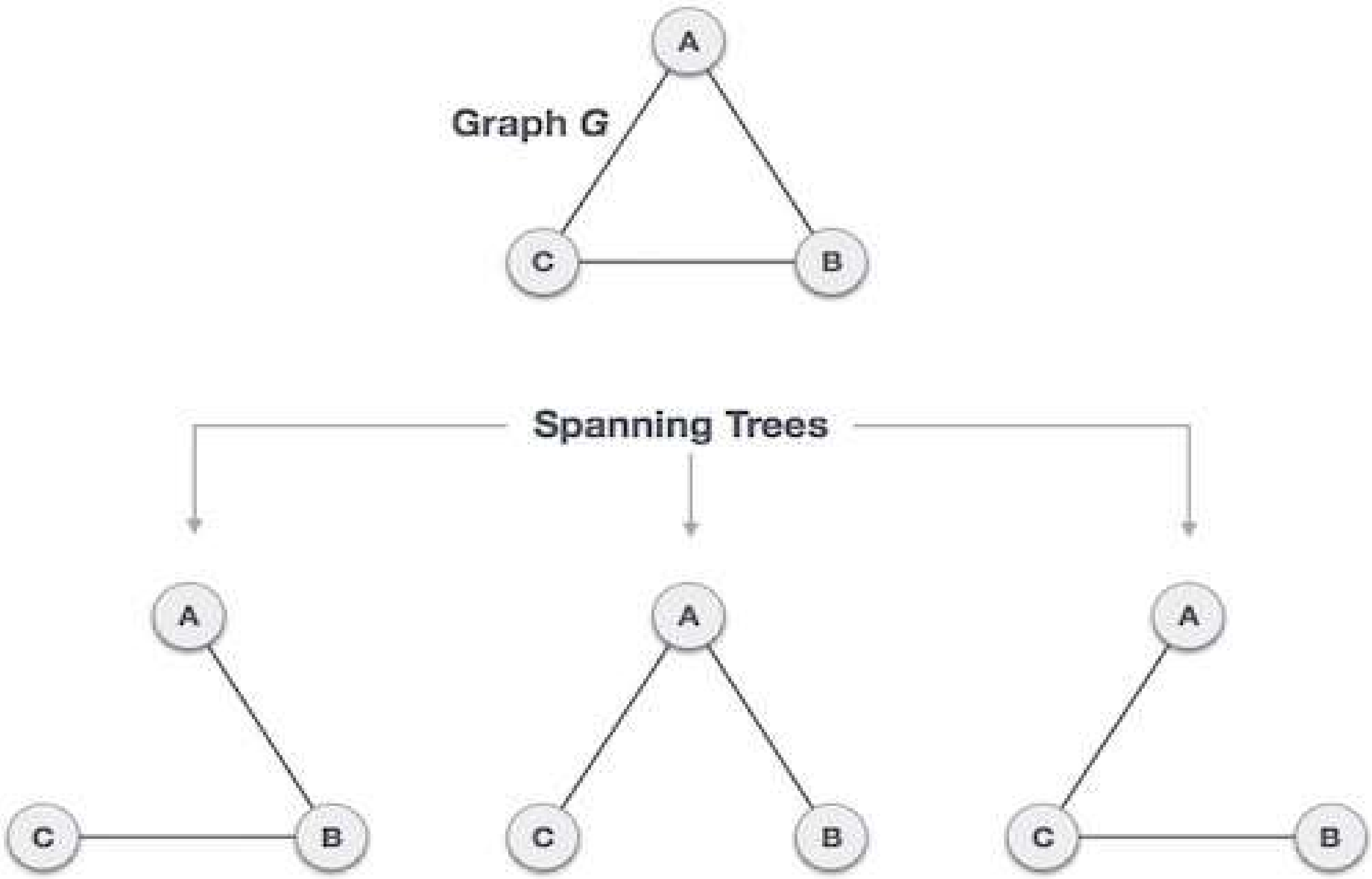


ST1(G)

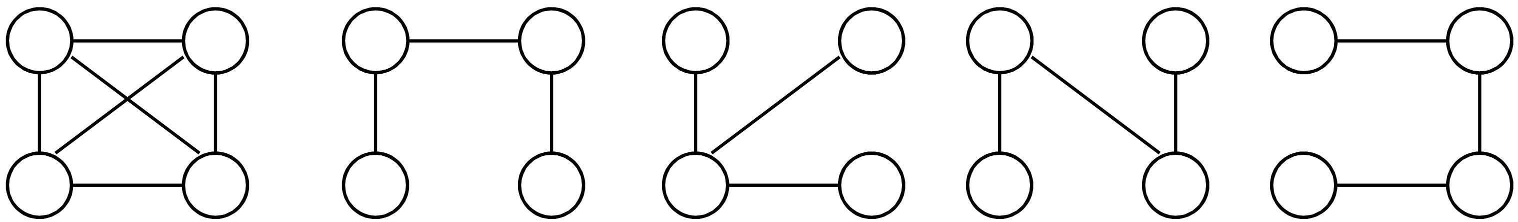


ST2(G)

Examples of spanning trees



•



A connected, undirected graph

Four of the spanning trees of the graph

CHAPTER 1

BASIC CONCEPT

All the programs in this file are selected from
Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed
"Fundamentals of Data Structures in C",
Computer Science Press, 1992.

Steps in system life cycle

- Requirements
- Analysis:
- Design: data objects and operations
- Refinement and Coding
- Verification
  - Program Proving
  - Testing
  - Debugging

Algorithm

- Definition
  An *algorithm* is a finite set of instructions that accomplishes a particular task.
- Criteria
  - input
  - output
  - definiteness: clear and unambiguous
  - finiteness: terminate after a finite number of steps
  - effectiveness: instruction is basic enough to be carried out

Performance Analysis and Measurements

- Performance Analysis (machine independent)
  - space complexity: storage requirement
  - time complexity: computing time
- Performance Measurement (machine dependent)

Space Complexity
$S(P)=C+S_P(I)$

- Memory space needed by a program:
  - Fixed Space requirements — C
  - Variable Space requirements — $S_p(I)$

Instance

if (true)

{ additional memory }

else

{ deleting the records }

Space Complexity
$S(P)=C+S_P(I)$

- Fixed Space Requirements (C)
  Independent of the characteristics of the inputs and outputs
  - instruction space
  - space for simple variables, fixed-size structured variable, constants
- Variable Space Requirements ($S_P(I)$)
  depend on the instance characteristic I
  - number, size, values of inputs and outputs associated with Instance
  - recursive stack space, formal parameters, local variables, return address

*Program 1.9: Simple arithmetic function (p.19)

```
float abc(float a, float b, float c)
{
   return a + b + b * c + (a + b - c) / (a + b) + 4.00;
 }
```

$\boxed{S_{abc}(I) = 0}$

*Program 1.10: Iterative function for summing a list of numbers (p.20)

```
float sum(float list[ ], int n)
{
  float tempsum = 0;
  int i;
  for (i = 0; i<n; i++)
    tempsum += list [i];
  return tempsum;
}
```

$\boxed{S_{sum}(I) = 0}$

Recall: pass the address of the first element of the array & pass by value

*Program 1.11: Recursive function for summing a list of numbers (p.20)

```
float rsum(float list[ ], int n)
{
  if (n) return rsum(list, n-1) + list[n-1];
  return 0;
}
```

$$\boxed{S_{sum}(I)=S_{sum}(n)=6n}$$

Assumptions:

*Figure 1.1: Space needed for one recursive call of Program 1.11 (p.21)

Type	Name	Number of bytes
parameter: float	list [ ]	2
parameter: integer	n	2
return address:(used internally)		2
TOTAL per recursive call		6

Time Complexity

- Time taken by a program : Compile time +Run Time

$$T(P)=C+T_P(I)$$

- Compile time is similar to the fixed space component
- Execution time depends on the program instances.
- For ex: Consider a simple program that adds and subtracts n numbers

$a = a + b$

$$T_P(n)=c_a ADD(n)+c_s SUB(n)+c_l LDA(n)+c_{st} STA(n)$$

- ca,cs,cl,cst are the constants that refer to the time needed to perform each operations: ADD,SUB, LOAD, STORE

Methods to compute the step count

- Introducing variable count into programs
- Tabular method
  - Determine the total number of steps contributed by each statement
    step per execution $\times$ frequency
  - add up the contribution of all statements

Time Complexity

- Time complexity computed by counting program steps

- Definition
  A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.

Regard as the same unit
machine independent

Iterative summing of a list of numbers

Step count method

*Program 1.12: Program 1.10 with count statements (p.23)

```
float sum(float list[ ], int n)
{
1   float tempsum = 0; count++; /* for assignment */
    int i;
n+1 for (i = 0; i < n; i++){
      count++;          /*for the for loop */
n     tempsum += list[i]; count++;  /* for assignment */
    }
    count++;        /* last execution of for */
    return tempsum;
1   count++;        /* for return */
}
```

$\boxed{2n + 3 \text{ steps}}$

Recursive summing of a list of numbers

Step count method

*Program 1.14: Program 1.11 with count statements added (p.24)

```
float rsum(float list[ ], int n)
{
count++;       /*for if conditional */
if (n) {
            count++;  /* for return and rsum invocation */
            return rsum(list, n-1) + list[n-1];
      }
      count++;
      return list[0];
}
```

$\boxed{2n+2}$

Matrix addition

Step count method

*Program 1.15: Matrix addition (p.25)

```
void add( int a[ ] [MAX_SIZE], int b[ ] [MAX_SIZE],
                    int c [ ] [MAX_SIZE], int rows, int cols)
{
   int i, j;
   for (i = 0; i < rows; i++)
     for (j= 0; j < cols; j++)
       c[i][j] = a[i][j] +b[i][j];
}
```

Step count method

*Program 1.16: Matrix addition with count statements (p.25)

rows+1

rows. (cols+1)

rows.cols

```
void add(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE],
                    int c[ ][MAX_SIZE], int row, int cols )
{
  int i, j;
  for (i = 0; i < rows; i++){
    count++; /* for i for loop */
    for (j = 0; j < cols; j++) {
      count++; /* for j for loop */
      c[i][j] = a[i][j] + b[i][j];
      count++; /* for assignment statement */
    }
    count++;    /* last time of j for loop */
  }
  count++;         /* last time of i for loop */
}
```

1 execu

$\boxed{2rows * cols + 2\ rows + 1}$

Step count method

*Program 1.17: Simplification of Program 1.16 (p.26)

```
void add(int a[ ][MAX_SIZE], int b [ ][MAX_SIZE],
                     int c[ ][MAX_SIZE], int rows, int cols)
{
   int i, j;
   for( i = 0; i < rows; i++) {
      for (j = 0; j < cols; j++)
         count += 2;
         count += 2;
   }
   count++;
}
```

$$2rows \times cols + 2rows + 1$$

Suggestion: Interchange the loops when rows >> cols

Tabular Method

*Figure 1.2: Step count table for Program 1.10 (p.26)

Iterative function to sum a list of numbers

steps/execution

Statement	s/e	Frequency	Total steps
float sum(float list[ ], int n)	0	0	0
{	0	0	0
float tempsum = 0;	1	1	1
int i;	0	0	0
for(i=0; i <n; i++)	1	n+1	n+1
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

Recursive Function to sum of a list of numbers

*Figure 1.3: Step count table for recursive summing function (p.27)

Statement	s/e	Frequency	Total steps
float rsum(float list[ ], int n)	0	0	0
{	0	0	0
if (n)	1	n+1	n+1
return rsum(list, n-1)+list[n-1];	1	n	n
return list[0];	1	1	1
}	0	0	0
Total			2n+2

Matrix Addition

*Figure 1.4: Step count table for matrix addition (p.27)

Statement	s/e	Frequency	Total steps
Void add (int a[ ][MAX_SIZE]. . . .)	0	0	0
{	0	0	0
int i, j;	0	0	0
for (i = 0; i < row; i++)	1	rows+1	rows+1
for (j=0; j< cols; j++)	1	rows. (cols+1)	rows. cols+rows
c[i][j] = a[i][j] + b[i][j];	1	rows. cols	rows. cols
}	0	0	0
Total			2rows. cols+2rows+1

Thank you