BCS202L- Data Structures and Algorithms

Dr. Iyappan Perumal
Assistant Professor Senior Grade 2
School of Computer Science & Engineering
VIT, Vellore.

BCS202L- Data Structures and Algorithms

- Module-1: Algorithm Analysis
- Module-2: Linear Data Structures
- Module-3: Searching and Sorting
- Module-4: Trees
- Module-5: Graphs
- Module-6: Hashing
- Module-7: Heaps and AVL Trees

Module:1	Algorithm Analysis	8 hours
Importance of algorithms and data structures - Fundamentals of algorithm analysis: Space and time complexity of an algorithm, Types of asymptotic notations and orders of growth - Algorithm efficiency – best case, worst case, average case - Analysis of non-recursive and recursive algorithms - Asymptotic analysis for recurrence relation: Iteration Method, Substitution Method, Master Method and Recursive Tree Method.		
Module:2	**Linear Data Structures**	**7 hours**
Arrays: 1D and 2D array- Stack - Applications of stack: Expression Evaluation, Conversion of Infix to postfix and prefix expression, Tower of Hanoi – Queue - Types of Queue: Circular Queue, Double Ended Queue (deQueue) - Applications – List: Singly linked lists, Doubly linked lists, Circular linked lists- Applications: Polynomial Manipulation.		
Module:3	**Searching and Sorting**	**7 hours**
Searching: Linear Search and binary search – Applications. Sorting: Insertion sort, Selection sort, Bubble sort, Counting sort, Quick sort, Merge sort - Analysis of sorting algorithms.		
Module:4	**Trees**	**6 hours**
Introduction - Binary Tree: Definition and Properties - Tree Traversals- Expression Trees:- Binary Search Trees - Operations in BST: insertion, deletion, finding min and max, finding the k^{th} minimum element.		
Module:5	**Graphs**	**6 hours**
Terminology – Representation of Graph – Graph Traversal: Breadth First Search (BFS), Depth First Search (DFS) - Minimum Spanning Tree: Prim's, Kruskal's - Single Source Shortest Path: Dijkstra's Algorithm.		
Module:6	**Hashing**	**4 hours**
Hash functions - Separate chaining - Open hashing: Linear probing, Quadratic probing, Double hashing - Closed hashing - Random probing – Rehashing - Extendible hashing.		
Module:7	**Heaps and AVL Trees**	**5 hours**
Heaps - Heap sort- Applications -Priority Queue using Heaps. AVL trees: Terminology, basic operations (rotation, insertion and deletion).		

BCS202L- Data Structures and Algorithms

Text Books:

1. Mark A. Weiss, Data Structures & Algorithm Analysis in C++, 4 th Edition, 2013,Pearson Education.

Reference Books:

1. Alfred V. Aho, Jeffrey D. Ullman and John E. Hopcroft, Data Structures and Algorithms, 1983, Pearson Education.
2. Horowitz, Sahni and S. Anderson-Freed, Fundamentals of Data Structures in C, 2008, 2nd Edition, Universities Press.
3. Thomas H. Cormen, C.E. Leiserson, R L. Rivest and C. Stein, Introduction to Algorithms, 2009, 3rd Edition, MIT Press.

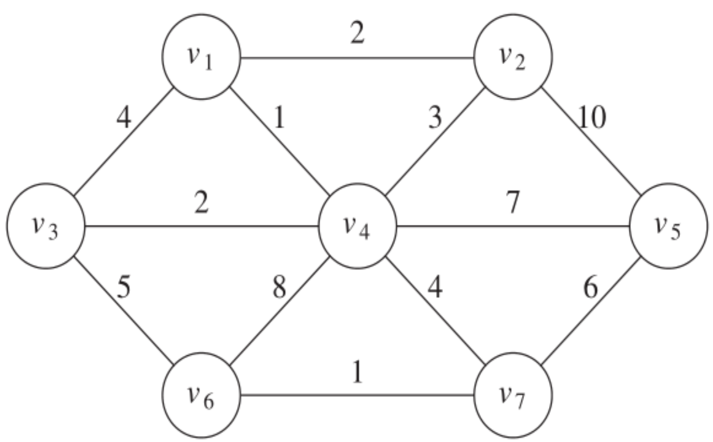
Module 5: Graphs(6 Hours)

- Terminology and Representation
- Graph Traversal
  - DFS
  - BFS
- **Minimum Spanning Tree**
  - **Prim's Algorithm**
  - **Kruskal's Algorithm**
- Single Source Shortest Path
  - Dijkstra's Algorithm

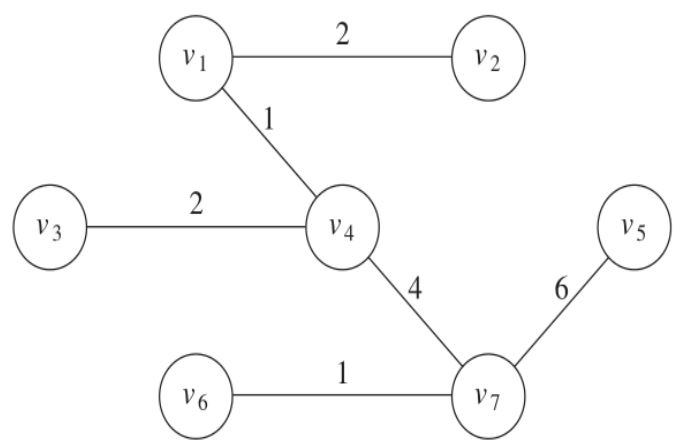
Minimum Spanning Tree

- A minimum spanning tree of an undirected graph G(V,E) is a **tree** formed from graph edges that connects all the vertices of G at lowest total cost.

G(V,E)



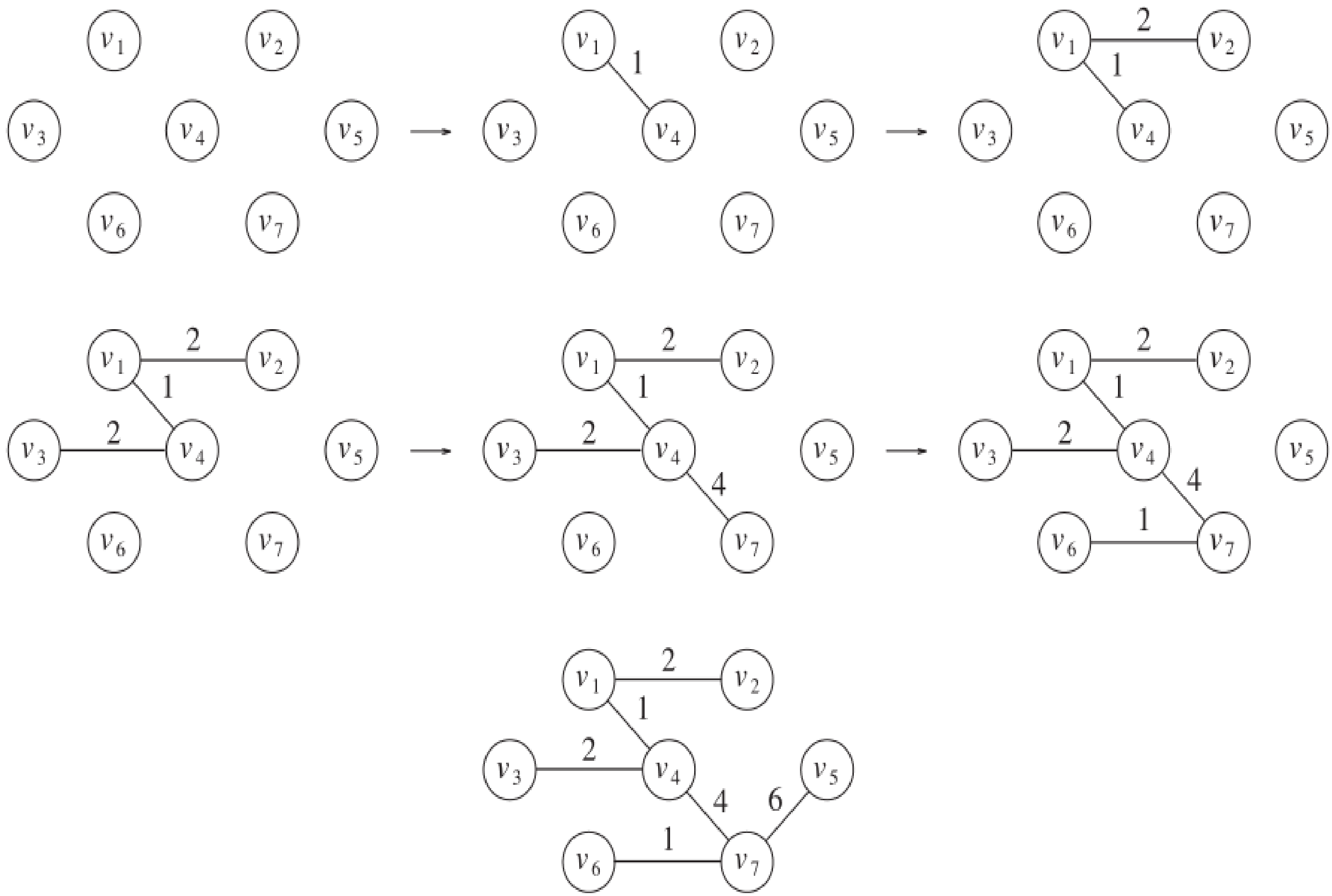
Minimum Spanning Tree



Prim's Algorithm

- Builds Tree edge by edge
- Next edge to include is chosen according to some Optimization Criteria
- Criteria??
  - **Choose an edge that results in a minimum increase in the sum of the costs of edges so far included.**

MST using Prim’s algorithm after each stage



Algorithm -Prim's algorithm

```
Algorithm Prim(E, cost, n, t)
// E is the set of edges in G. cost[1 : n, 1 : n] is the cost
// adjacency matrix of an n vertex graph such that cost[i, j] is
// either a positive real number or ∞ if no edge (i, j) exists.
// A minimum spanning tree is computed and stored as a set of
// edges in the array t[1 : n − 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge in
// the minimum-cost spanning tree. The final cost is returned.
{
    Let (k, l) be an edge of minimum cost in E;
    mincost := cost[k, l];
    t[1, 1] := k; t[1, 2] := l;
    for i := 1 to n do // Initialize near.
        if (cost[i, l] < cost[i, k]) then near[i] := l;
        else near[i] := k;
    near[k] := near[l] := 0;
    for i := 2 to n − 1 do
    { // Find n − 2 additional edges for t.
        Let j be an index such that near[j] ≠ 0 and
        cost[j, near[j]] is minimum;
        t[i, 1] := j; t[i, 2] := near[j];
        mincost := mincost + cost[j, near[j]];
        near[j] := 0;
        for k := 1 to n do // Update near[ ].
            if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
                then near[k] := j;
    }
    return mincost;
}
```

PART 1

Destination

Source

PART 3

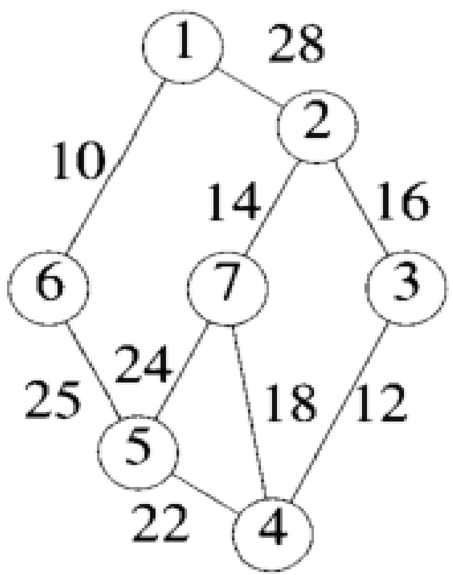
PART 4

```
Min=∞;
For k=1 to n do
{
    if(near[k]!=0)
        if[cost[k][near[k]]<min
        {
            min=cost[k][near[k]];
            j=k;
        }
}
```

PART 2

With respect to newly added edge update the near

MST using Prim's algorithm



Graph G={V,E)

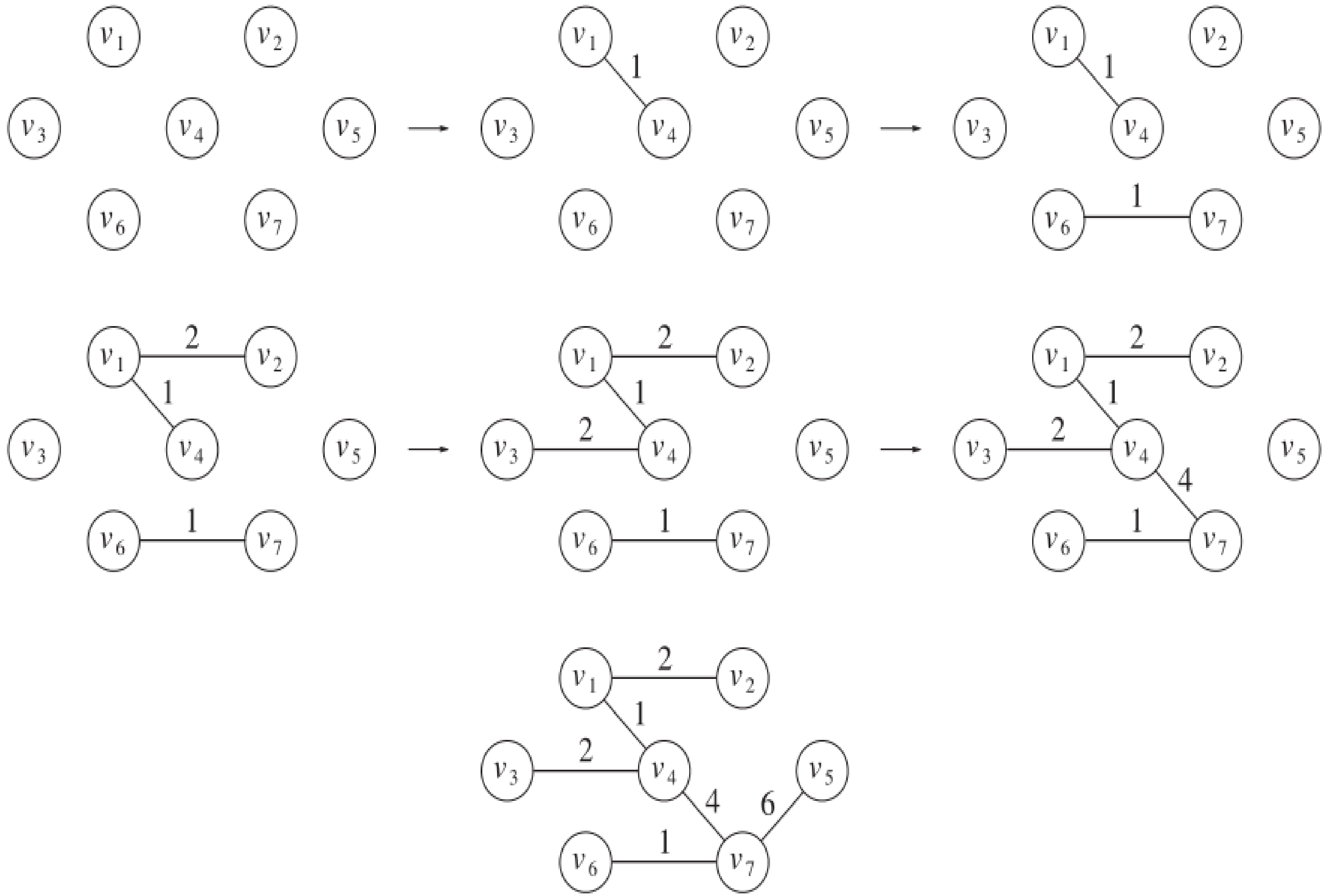
	1	2	3	4	5	6	7
1	∞	28	∞	∞	∞	10	∞
2	28	∞	16	∞	∞	∞	14
3	∞	16	∞	12	∞	∞	∞
4	∞	∞	12	∞	22	∞	18
5	∞	∞	∞	22	∞	25	24
6	10	∞	∞	∞	25	∞	∞
7	∞	14	∞	18	24	∞	∞

Adjacency Matrix

Kruskal's Algorithm

- Edges of the graph are arranged in increasing order of cost.
- Continuously select the edges in order of smallest weight and accept an edge if it does not cause a cycle.

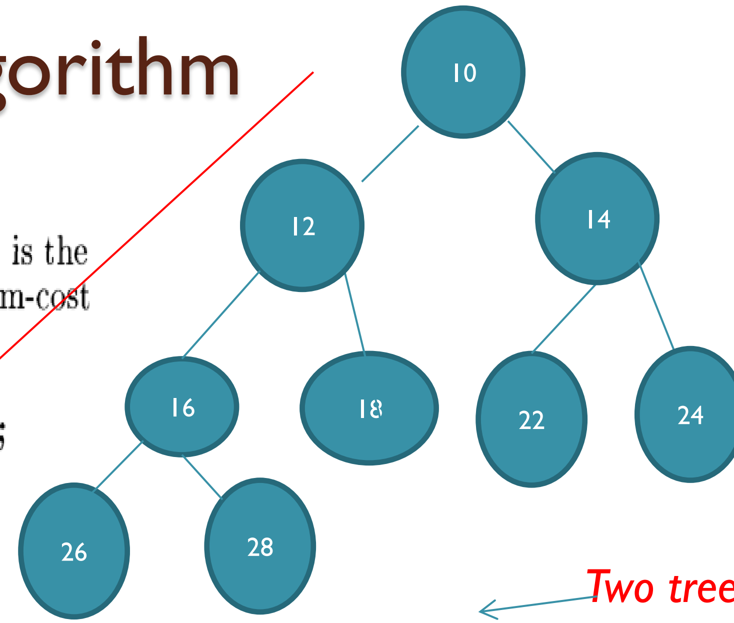
MST using Kruskal's algorithm after each stage



Algorithm –Kruskal's algorithm

```
Algorithm Kruskal(E, cost, n, t)
// E is the set of edges in G. G has n vertices. cost[u, v] is the
// cost of edge (u, v). t is the set of edges in the minimum-cost
// spanning tree. The final cost is returned.
{
    Construct a heap out of the edge costs using Heapify;
    for i := 1 to n do parent[i] := -1;
    // Each vertex is in a different set.
    i := 0; mincost := 0.0;
    while ((i < n - 1) and (heap not empty)) do
    {
        Delete a minimum cost edge (u, v) from the heap
        and reheapify using Adjust;
        j := Find(u); k := Find(v);
        if (j ≠ k) then
        {
            i := i + 1;
            t[i, 1] := u; t[i, 2] := v;
            mincost := mincost + cost[u, v];
            Union(j, k);
        }
    }
    if (i ≠ n - 1) then write ("No spanning tree");
    else return mincost;
}
```

Checking for cycle



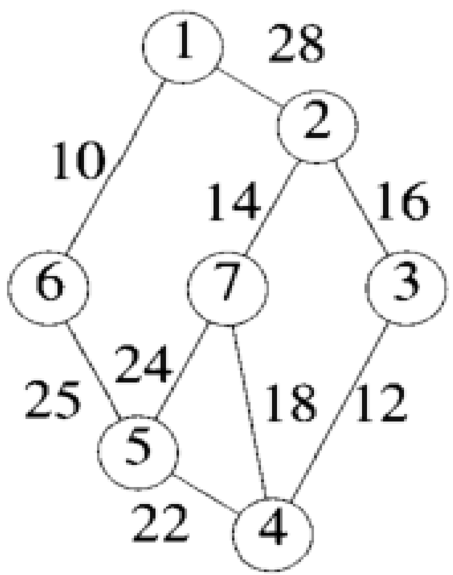
Two trees with roots i and j should be joined

```
Algorithm SimpleUnion(i, j)
{
    p[i] := j;
}
```

```
Algorithm SimpleFind(i)
{
    while (p[i] ≥ 0) do i := p[i];
    return i;
}
```

Determines the Root of the tree containing element i

MST using Kruskal's algorithm



Graph G={V,E)

	1	2	3	4	5	6	7
1	∞	28	∞	∞	∞	10	∞
2	28	∞	16	∞	∞	∞	14
3	∞	16	∞	12	∞	∞	∞
4	∞	∞	12	∞	22	∞	18
5	∞	∞	∞	22	∞	25	24
6	10	∞	∞	∞	25	∞	∞
7	∞	14	∞	18	24	∞	∞

Adjacency Matrix