CMPE 180-92

Data Structures and Algorithms in C++

November 30 Class Meeting

Department of Computer Engineering
San Jose State University

Fall 2017

Instructor: Ron Mak

www.cs.sjsu.edu/~mak

Assignment #12: Extra Credit

- Modify file SortTests.cpp to record the sorting statistics in instances of this struct:

```
struct Stats
{
    long move_count;
    long compare_count;
    long elapsed_ms;
};
```

Assignment #12: Extra Credit, *cont'd*

- Store the struct instances in a "three dimensional map":

```
typedef map<string, map<int, map<string, Stats>>> StatsMap;
```

Data generator name
"Unsorted random"
"Already sorted"
"Reverse sorted"
"All zeros"

Data size

Sorting algorithm name
"Selection sort"
"Insertion sort"
"Shellsort suboptimal"
"Shellsort optimal"
"Quicksort suboptimal"
"Quicksort optimal"
"Mergesort"

```
Stats stats;
StatsMap stats_map;
...
stats_map[generator_name][n][sorter->name()] = stats;
```

Assignment #12: Extra Credit, *cont'd*

- Run the sorting algorithms with data size N = 10,000 to 100,000 by increments of 10,000.

- Output the stored stats as comma-separated values (CSV) and copy into text file stats.csv with column headers. Open the file with Excel.

```
Moves: Unsorted random
N,Selection,Insertion,Shell Sub,Shell Opt,Quick Sub,Quick Opt, Merge
10000,19980,24743369,209775,208781,91704,104618,150526
20000,39978,99248464,496880,512315,190888,218234,320901
30000,59986,223662787,745760,792562,293236,334760,498479
40000,79982,402835573,1160168,1099597,400818,454202,681667
50000,99970,627376797,1559713,1493082,509092,572150,868016
60000,119986,902496559,1760544,2068466,620352,695636,1057385
70000,139976,1221025827,2192555,2372181,724066,818264,1248891
80000,159974,1603609322,2729890,2543970,846812,943912,1443544
90000,179984,2021919155,3182741,3208782,960306,1073462,1639240
100000,199980,2502725426,3806995,3505466,1056514,1194464,1836312
```

How to Chart with Excel

- In Excel, select the values you want to graph and use the main menu:
  Insert ➔ Chart ➔ X Y (Scatter)

- Right-click the graph: Select Data ...

- Click on a dot and right-click: Add Trendline
  - Select Polynomial with order 2 (or higher).

- Save the spreadsheet as an Excel Workbook with suffix .xlsx
  - Otherwise, you lose the graphs.

Assignment #13 Part 1 Solution

- Case 1 (outside left-left):
  Rebalance with a single right rotation.

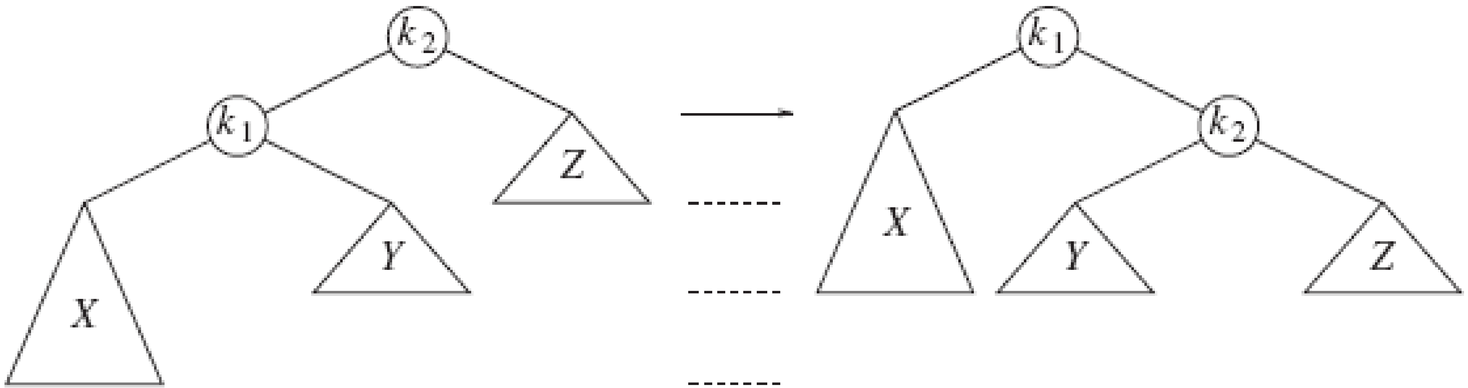


Figure 4.31 Single rotation to fix case 1

Assignment #13 Part 1 Solution, *cont'd*

```
/**
 * Case 1 (outside left-left): Rebalance with a single right rotation.
 * Update heights and return the new root node.
 * @param k2 pointer to the node to rotate.
 * @return pointer to the new root node.
 */
BinaryNode *AvlTree::singleRightRotation(BinaryNode *k2)
{
    BinaryNode *k1 = k2->left;

    // Rotate.
    k2->left = k1->right;
    k1->right = k2;

    // Recompute node heights.
    k2->height = (max(height(k2->left), height(k2->right)) + 1);
    k1->height = (max(height(k1->left), k2->height) + 1);

    return k1;
}
```

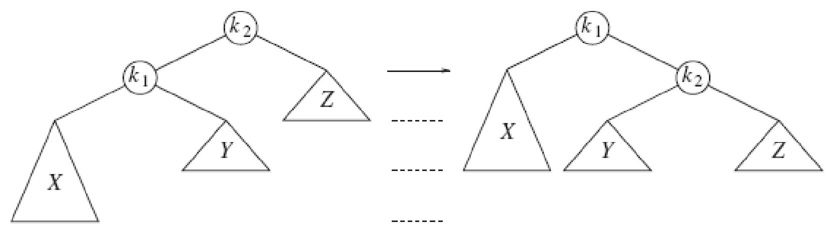


Figure 4.31 Single rotation to fix case 1

Assignment #13 Part 1 Solution, *cont'd*

- Case 4 (outside right-right):
  Rebalance with a single left rotation.

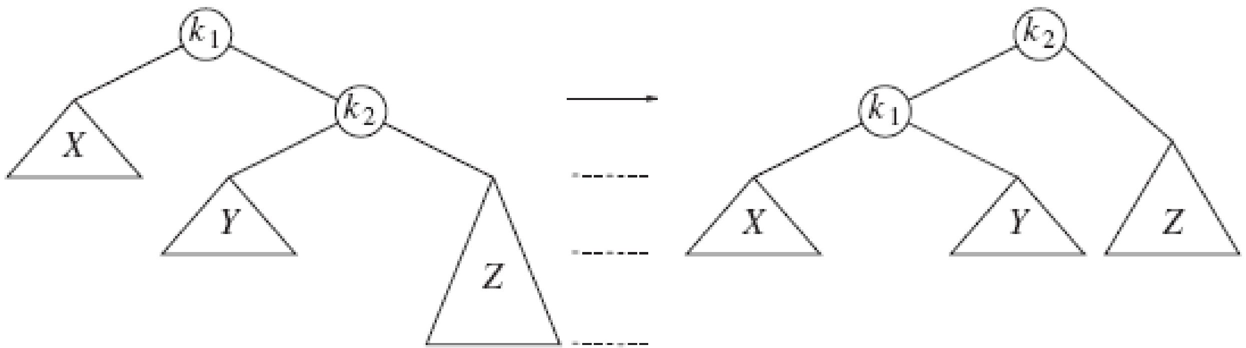


Figure 4.33 Single rotation fixes case 4

Data Structures and Algorithm Analysis in C++, 4th ed.
by Mark Allen Weiss
Pearson Education, Inc., 2014

Assignment #13 Part 1 Solution, *cont'd*

```
/**
 * Case 4 (outside right-right): Rebalance with a single left rotation.
 * Update heights and return the new root node.
 * @param k2 pointer to the node to rotate.
 * @return pointer to the new root node.
 */
BinaryNode *AvlTree::singleLeftRotation(BinaryNode *k1)
{
    BinaryNode *k2 = k1->right;

    // Rotate.
    k1->right = k2->left;
    k2->left  = k1;

    // Recompute node heights.
    k1->height = (max(height(k1->left), height(k1->right)) + 1);
    k2->height = (max(height(k2->right), k1->height) + 1);

    return k2;
}
```

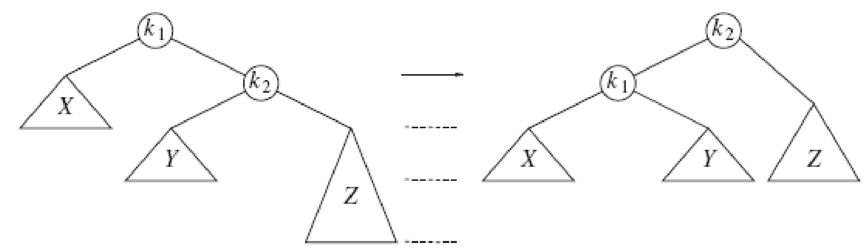


Figure 4.33 Single rotation fixes case 4

Assignment #13 Part 1 Solution, *cont'd*

- Case 2 (inside left-right):
  Rebalance with a double left-right rotation.

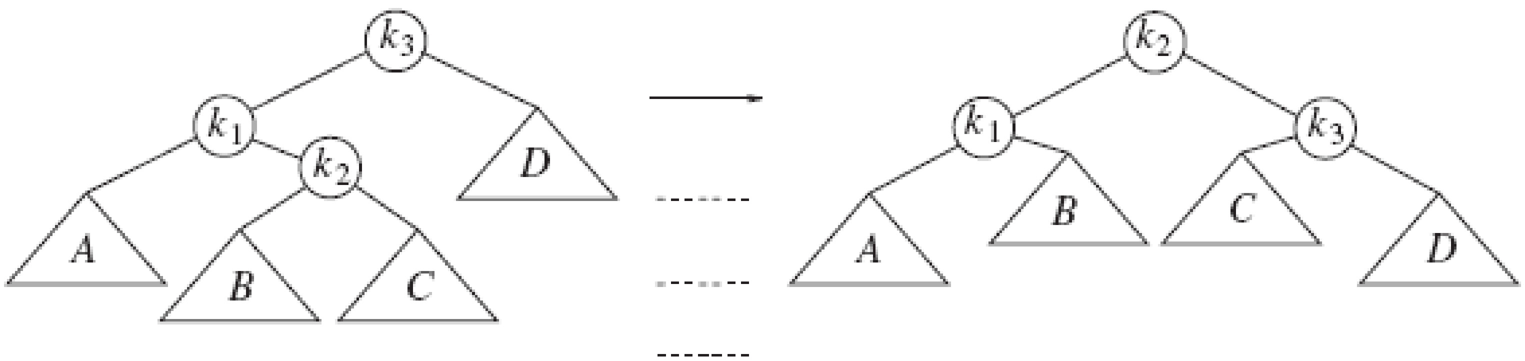


Figure 4.35 Left–right double rotation to fix case 2

Assignment #13 Part 1 Solution, *cont'd*

```
/**
 * Case 2 (inside left-right): Rebalance with a double left-right rotation.
 * @param k3 pointer to the node to rotate.
 * @return pointer to the new root node.
 */
BinaryNode *AvlTree::doubleLeftRightRotation(BinaryNode *k3)
{
    k3->left = singleLeftRotation(k3->left);
    return singleRightRotation(k3);
}
```

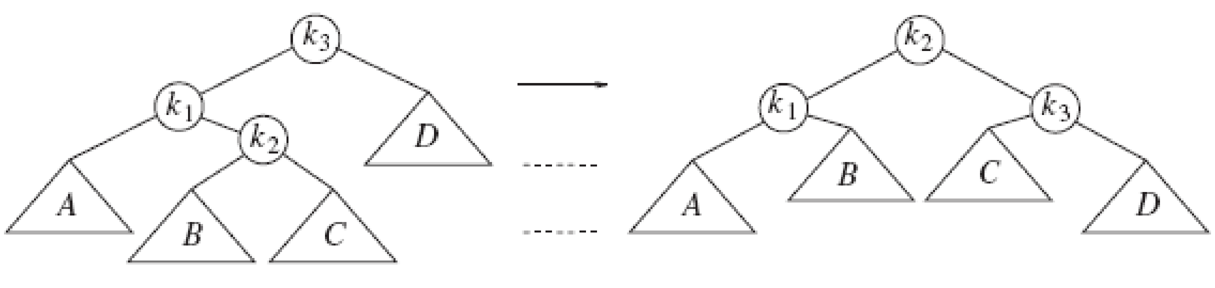


Figure 4.35 Left–right double rotation to fix case 2

Assignment #13 Part 1 Solution, *cont'd*

- Case 3 (inside right-left):
  Rebalance with a double right-left rotation.

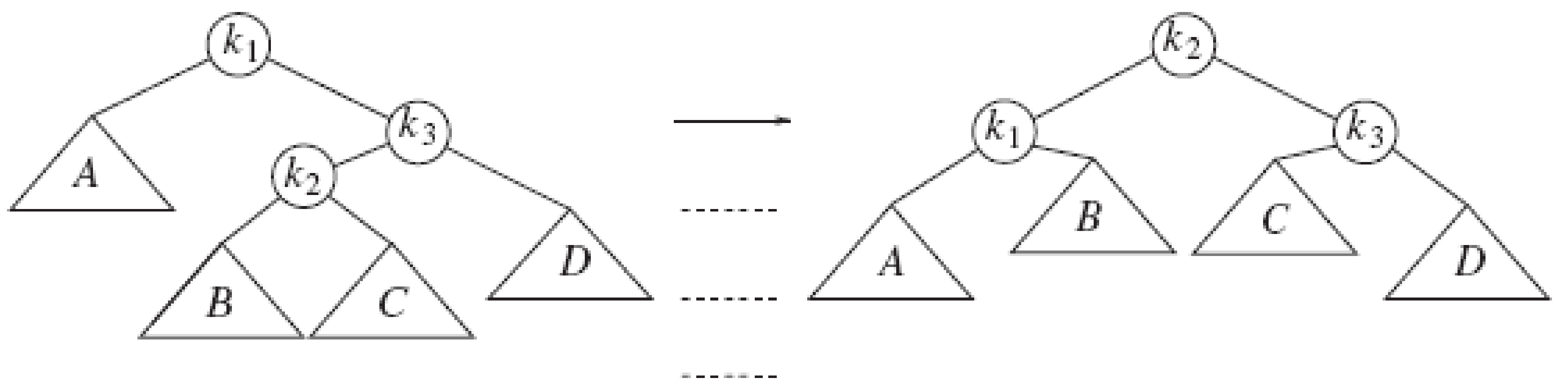


Figure 4.36 Right–left double rotation to fix case 3

Assignment #13 Part 1 Solution, *cont'd*

```
/**
 * Case 3 (inside right-left): Rebalance with a double left rotation.
 * @param k1 pointer to the node to rotate.
 * @return pointer to the new root node.
 */
BinaryNode *AvlTree::doubleRightLeftRotation(BinaryNode *k1)
{
    k1->right = singleRightRotation(k1->right);
    return singleLeftRotation(k1);
}
```

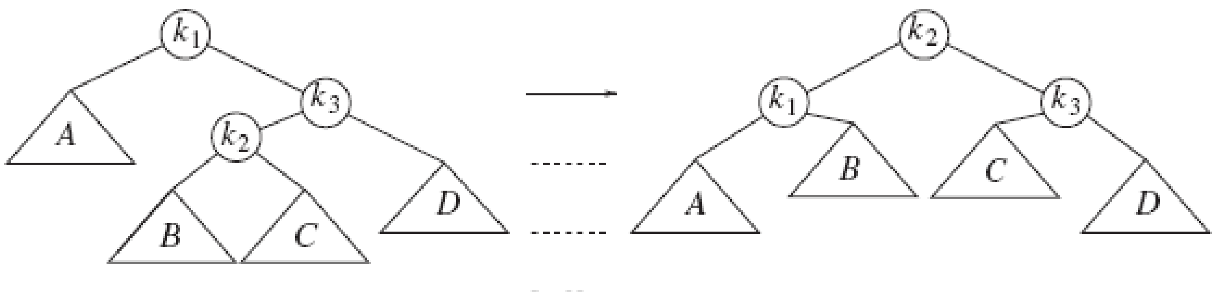
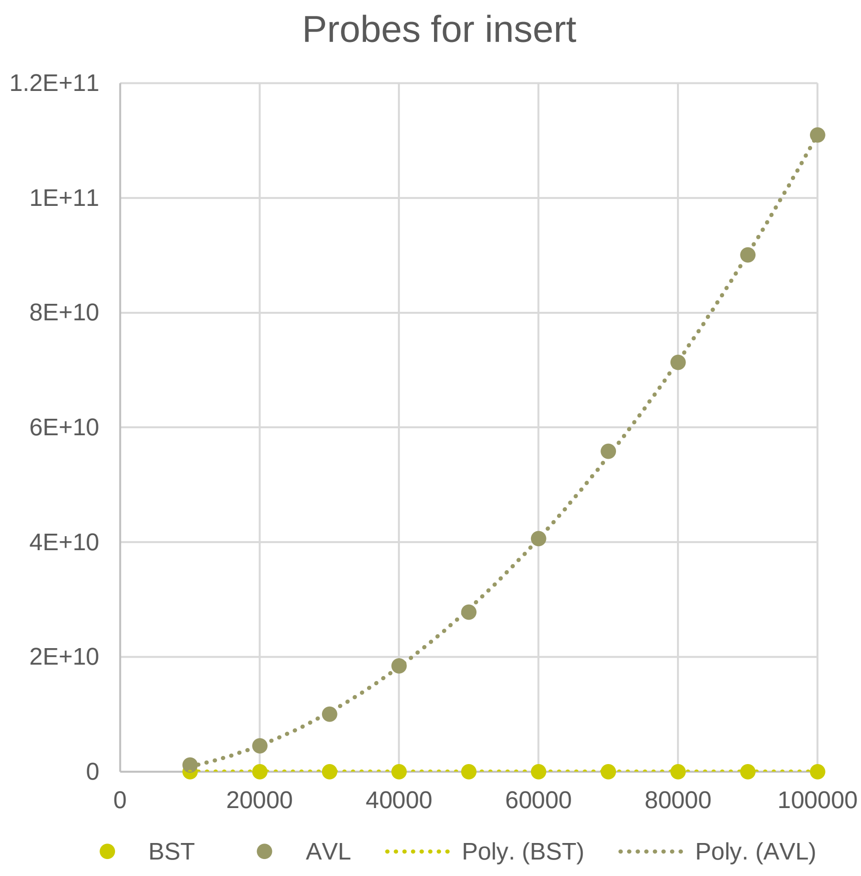
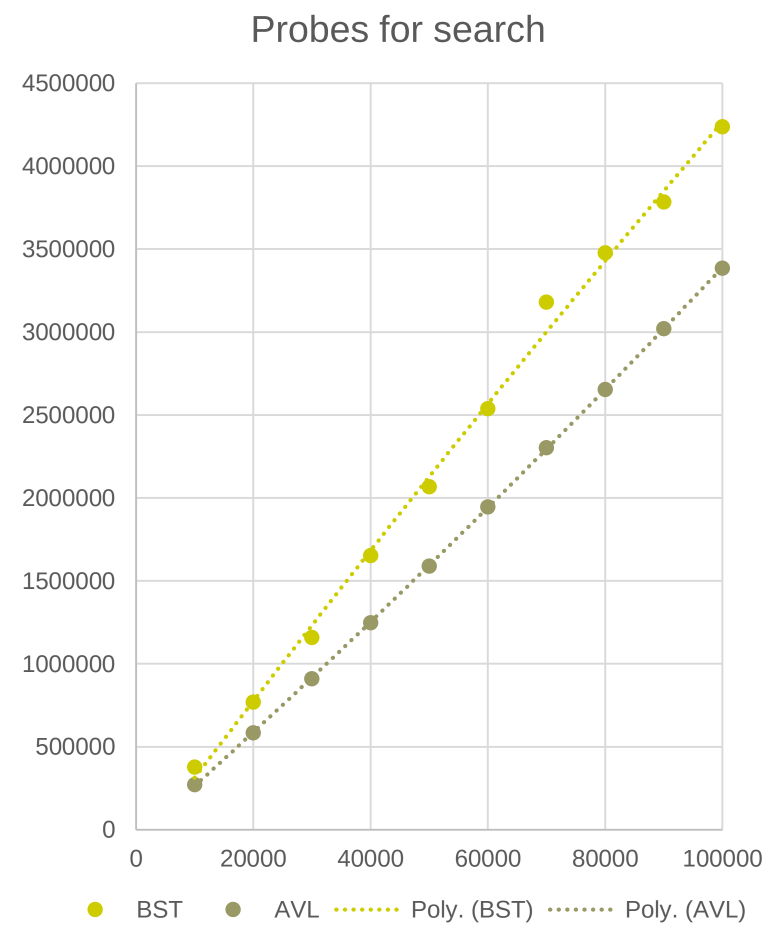


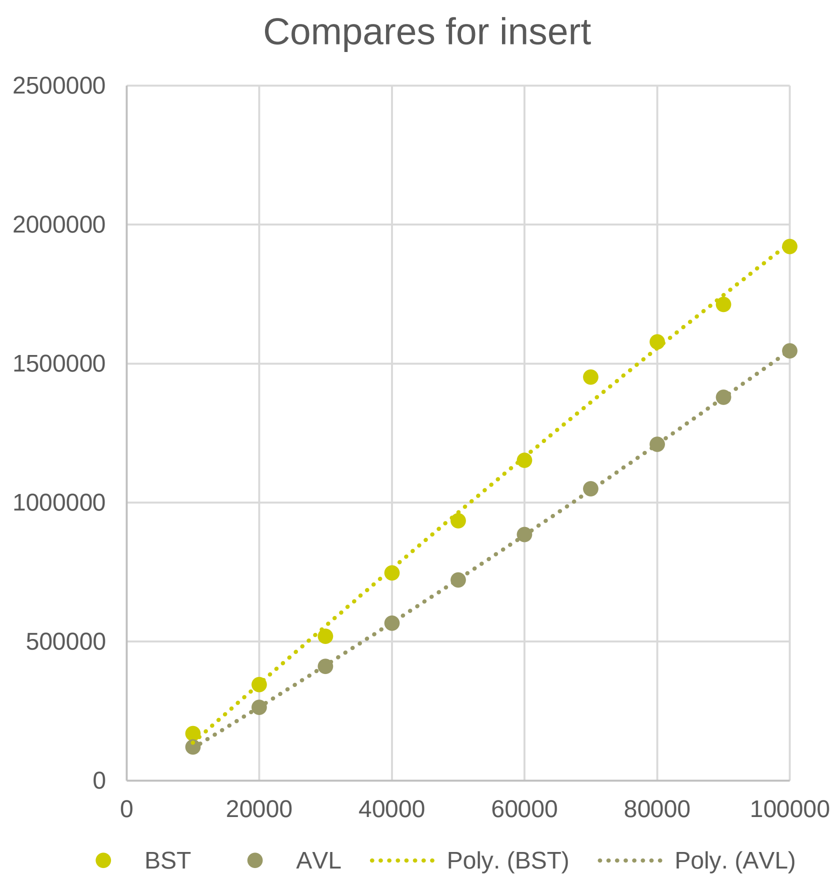
Figure 4.36 Right–left double rotation to fix case 3

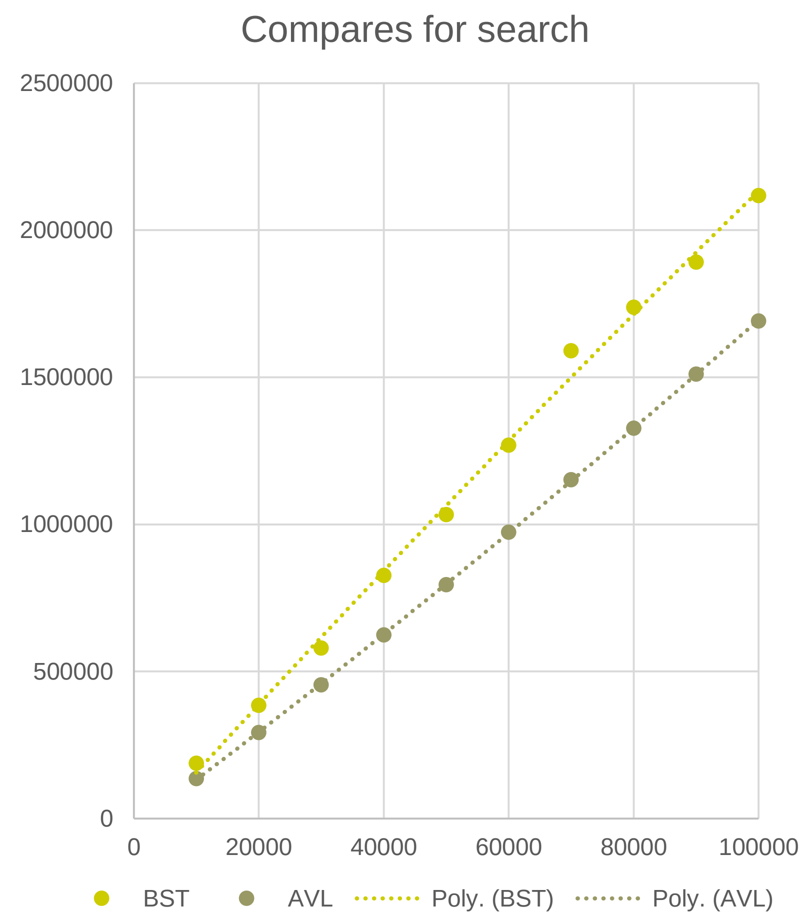
Assignment #13 Part 2 Solution



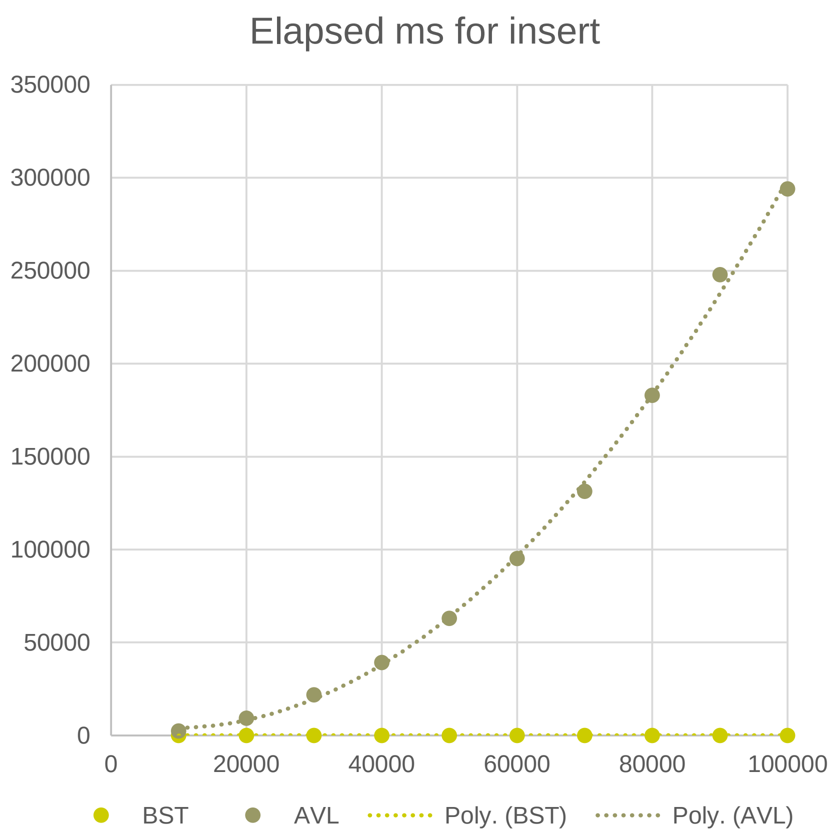


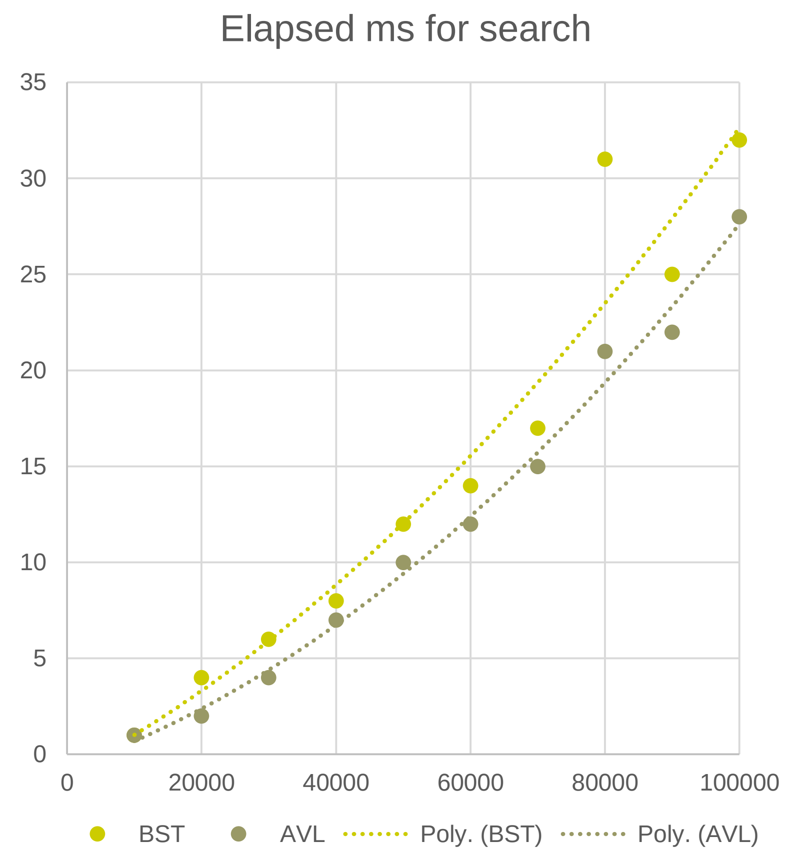
Assignment #13 Part 2 Solution, *cont'd*





Assignment #13 Part 2 Solution, *cont'd*





Graphs

- A graph is one of the most versatile data structures in computer science.

Uses of Graphs

- Model connectivity in computer and communications networks.
- Represent a map of locations and distances between them.
- Model flow capacities in transportation networks.
- Find a path from a starting condition to a goal condition.
- Model state transitions in computer algorithms.
- Model an order for finishing subtasks in a complex activity.
- Model relationships such as family trees, business and military organizations, and scientific taxonomies.

Graph Terms

- A graph $G = (V, E)$ is a set of vertices V and a set of edges (arcs) E.
- An edge is a pair (v, w), where v and w are in V.
- If the pair is ordered, the graph is directed and is called a digraph.

Graph Terms, *cont'd*

- Vertex *w* is adjacent to vertex *v* if and only if (*v*, *w*) is in *E.*

- In an undirected graph, both (*v*, w) and (*w*, *v*) are in *E*.
  - *v* is adjacent to *w*, and *w* is adjacent to *v*.

- An edge can have a weight or cost component.

Graph Examples

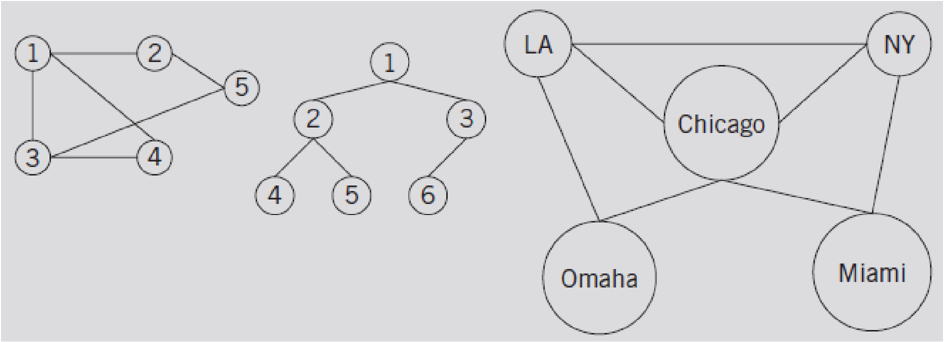


FIGURE 12-3 Various undirected graphs

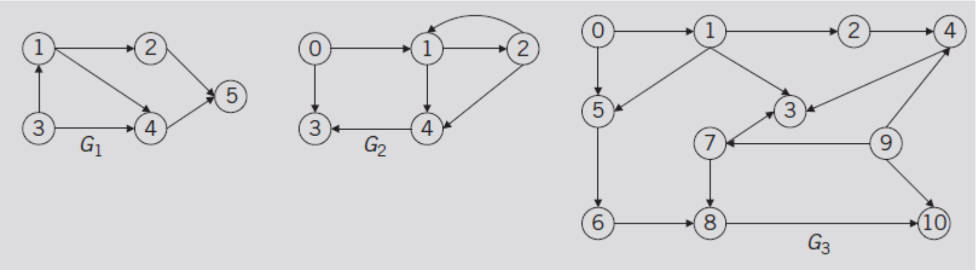


FIGURE 12-4 Various directed graphs

Data Structures Using C++
by D.S. Malik
Course Technology, 2010

Graph Terms, *cont'd*

- A **path** is a sequence of vertices w_1, w_2, w_3, ..., w_N where (w_i, w_{i+1}) is in E, for $1 \le i < N$.
- The **length** of the path is the number of edges on the path.
- A **simple path** has all distinct vertices, except that the first and last can be the same.

Graph Terms, *cont'd*

- A cycle in a directed graph is a path of length ≥ 1 where $w_1 = w_N$.
- A directed graph with no cycles is acyclic.
- A DAG is a directed acyclic graph.

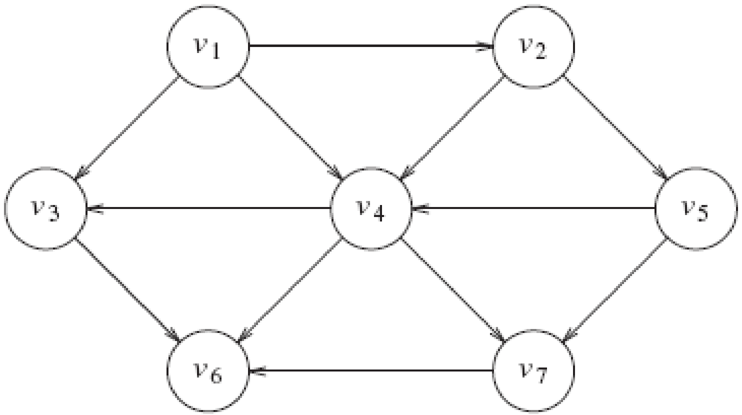


Figure 9.4 An acyclic graph

Graph Terms, *cont'd*

- The indegree of a vertex *v* is the number of incoming edges (*u*, *v*).

Graph Representation

- Represent a directed graph with an adjacency list.
  - For each vertex, keep a list of all adjacent vertices.

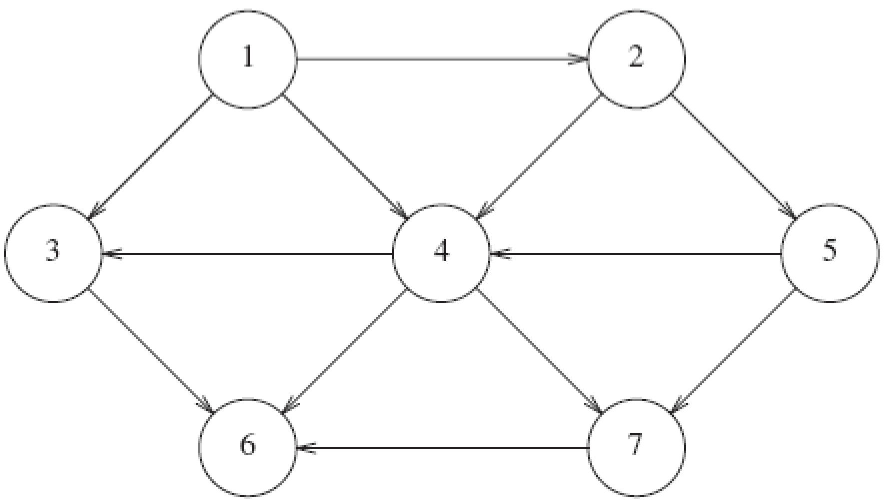


Figure 9.1 A directed graph

1	2, 4, 3
2	4, 5
3	6
4	6, 7, 3
5	4, 7
6	(empty)
7	6

Figure 9.2 An adjacency list representation of a graph

Topological Sort

- We can use a graph to represent the <u>prerequisites</u> in a course of study.
  - A directed edge from Course A to Course B means that Course A is a prerequisite for Course B.

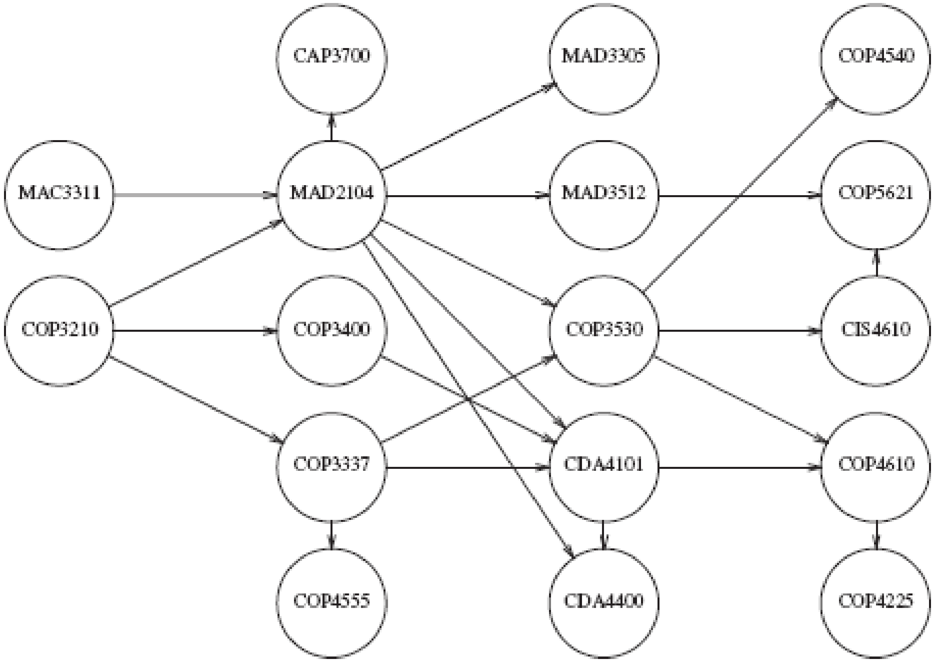


Figure 9.3 An acyclic graph representing course prerequisite structure

Topological Sort, *cont'd*

- A topological sort of a directed graph is an ordering of the vertices such that if there is a path from v_i to v_j, then v_i comes before v_j in the ordering.
  - The order is not necessarily unique.

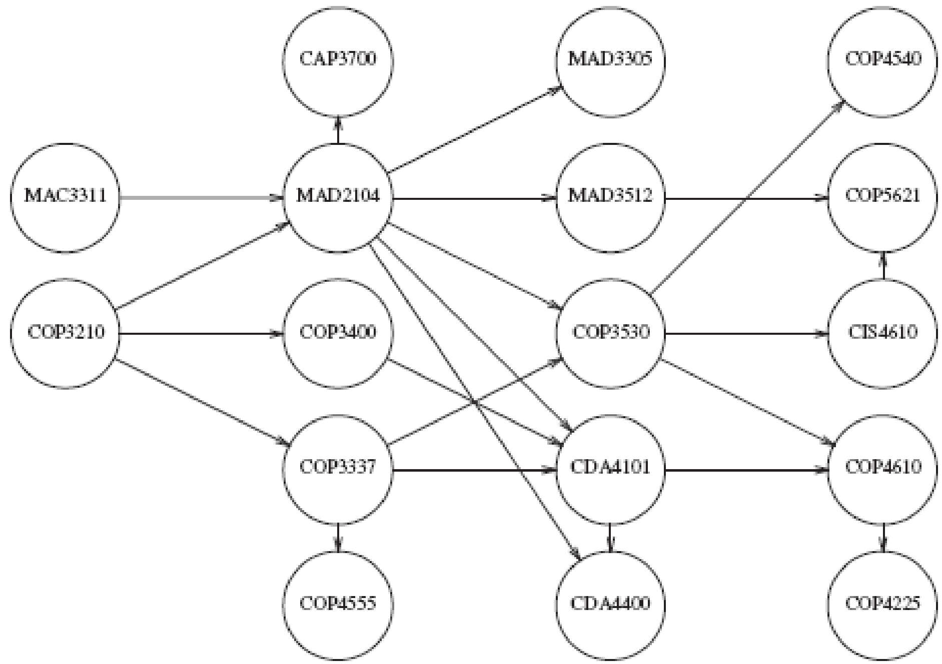


Figure 9.3 An acyclic graph representing course prerequisite structure

- Topological sort example using a queue.
  - Start with vertex v_1.
  - On each pass, remove the vertices with indegree = 0.
    - Subtract 1 from the indegree of the adjacent vertices.

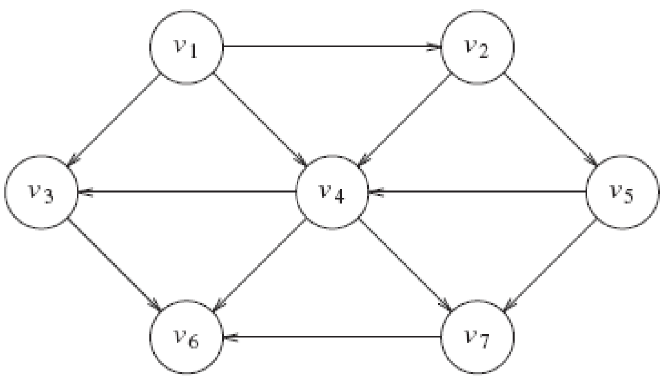


Figure 9.4 An acyclic graph

The topological sort is the order in which the vertices dequeue.

	Indegree Before Dequeue #						
Vertex	1	2	3	4	5	6	7
v_1	0	0	0	0	0	0	0
v_2	1	0	0	0	0	0	0
v_3	2	1	1	1	0	0	0
v_4	3	2	1	0	0	0	0
v_5	1	1	0	0	0	0	0
v_6	3	3	3	3	2	1	0
v_7	2	2	2	1	0	0	0
Enqueue	v_1	v_2	v_5	v_4	v_3, v_7		v_6
Dequeue	v_1	v_2	v_5	v_4	v_3	v_7	v_6

Figure 9.6 Result of applying topological sort to the graph in Figure 9.4

Topological Sort, *cont'd*

- Pseudocode to perform a topological sort.
  - $O(|E| + |V|)$ time

```
void topsort( ) throws CycleFoundException
{
    Queue<Vertex> q = new Queue<Vertex>( );
    int counter = 0;

    for each Vertex v
        if( v.indegree == 0 )
            q.enqueue( v );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );
        v.topNum = ++counter;  // Assign next number

        for each Vertex w adjacent to v
            if( --w.indegree == 0 )
                q.enqueue( w );
    }
    if( counter != NUM_VERTICES )
        throw new CycleFoundException( );
}
```

Figure 9.7 Pseudocode to perform topological sort

Shortest Path Algorithms

- Assume there is a cost associated with each edge.
  - The cost of a path is the sum of the cost of each edge on the path.
- Find the least-cost path from a “distinguished” vertex *s* to every other vertex in the graph.

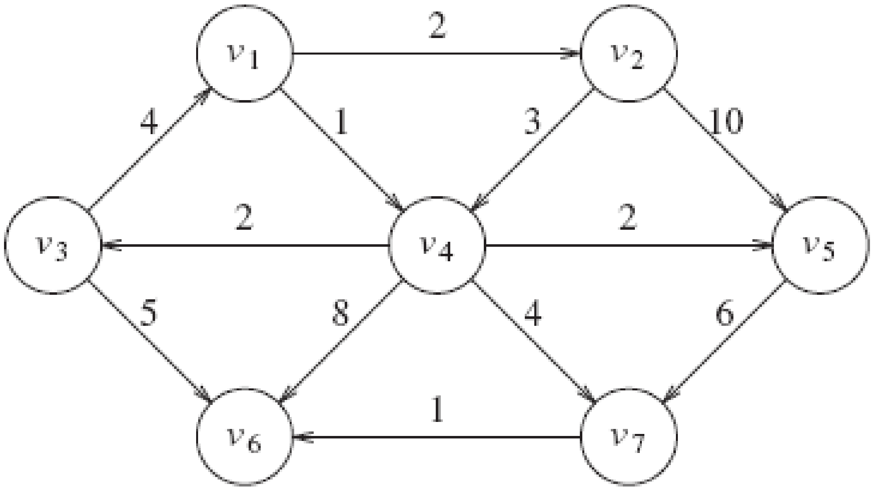


Figure 9.8 A directed graph G

Data Structures and Algorithm Analysis in C++, 4th ed.
by Mark Allen Weiss
Pearson Education, Inc., 2014

Shortest Path Algorithms, *cont'd*

- A negative cost results in a negative-cost cycle.

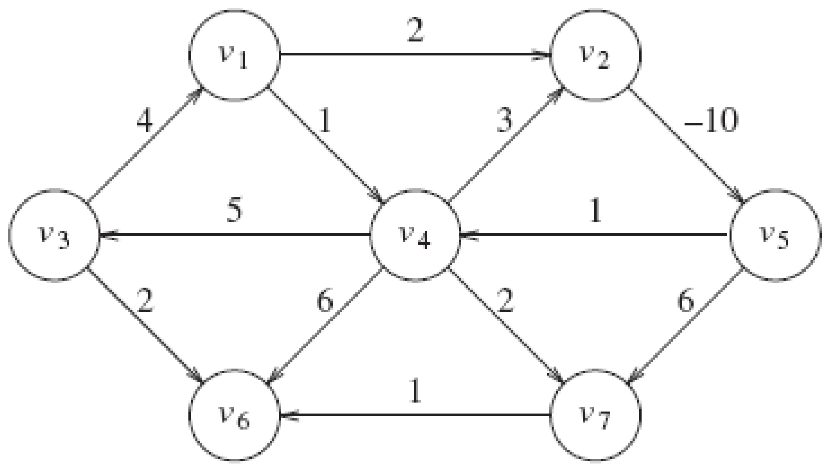


Figure 9.9 A graph with a negative-cost cycle

  - Make a path's cost <u>arbitrarily small</u> by looping.

Unweighted Shortest Path

- Minimize the lengths of paths.
  - Assign a weight of 1 to each edge.

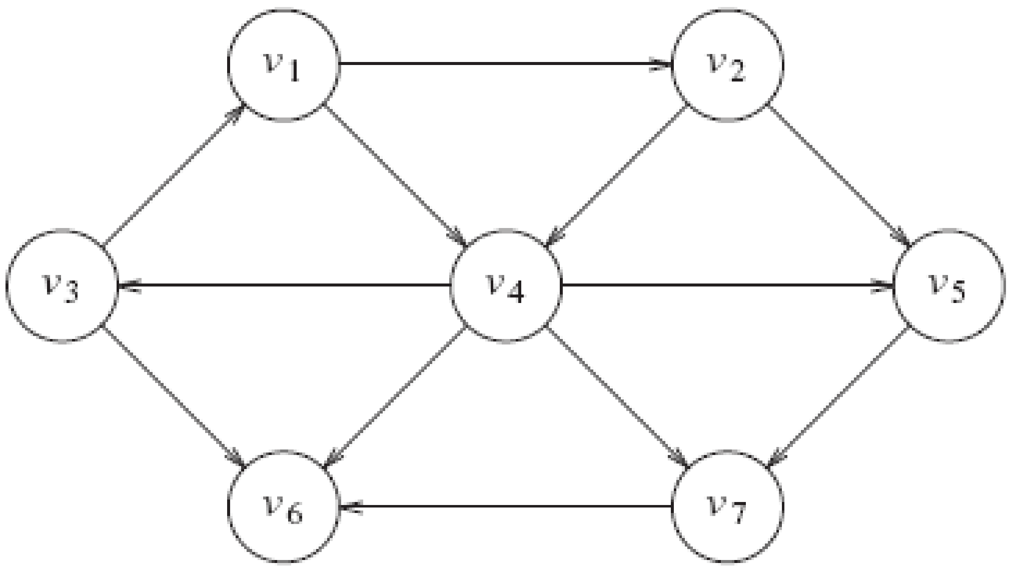


Figure 9.10 An unweighted directed graph G

  - In this example, let the distinguished vertex *s* be v_3.

Data Structures and Algorithm Analysis in C++, 4th ed.
by Mark Allen Weiss
Pearson Education, Inc., 2014

Unweighted Shortest Path, *cont'd*

- The path from *s* to itself has length (cost) 0.

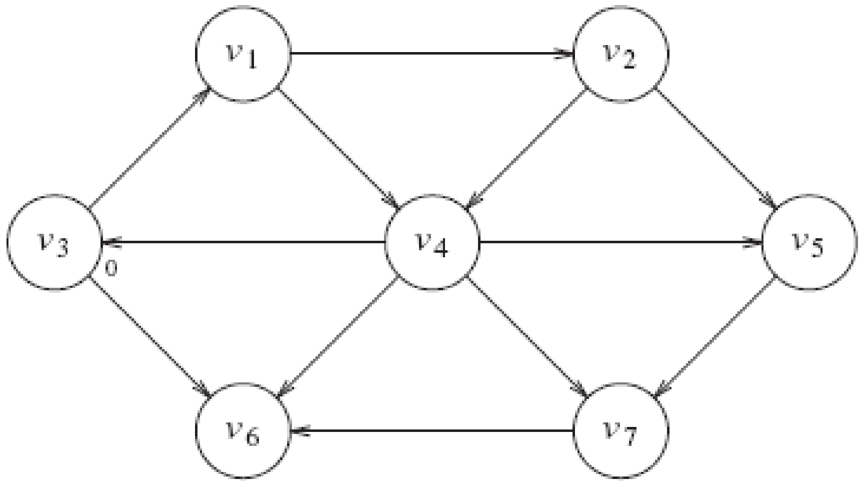


Figure 9.11 Graph after marking the start node as reachable in zero edges

Unweighted Shortest Path, *cont'd*

- Find vertices v_1 and v_6 that are distance 1 from v_3:

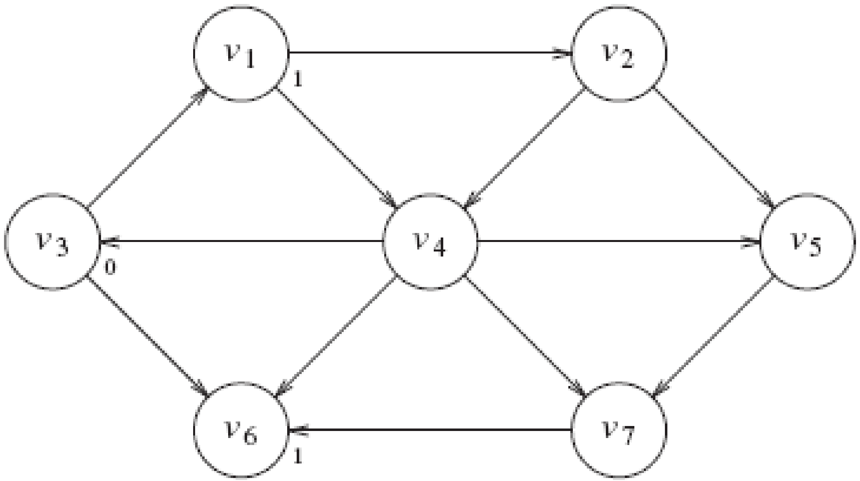


Figure 9.12 Graph after finding all vertices whose path length from *s* is 1

Unweighted Shortest Path, *cont'd*

- Find all vertices that are distance 2 from v_3.
  - Begin with the vertices adjacent to v_1 and v_6.

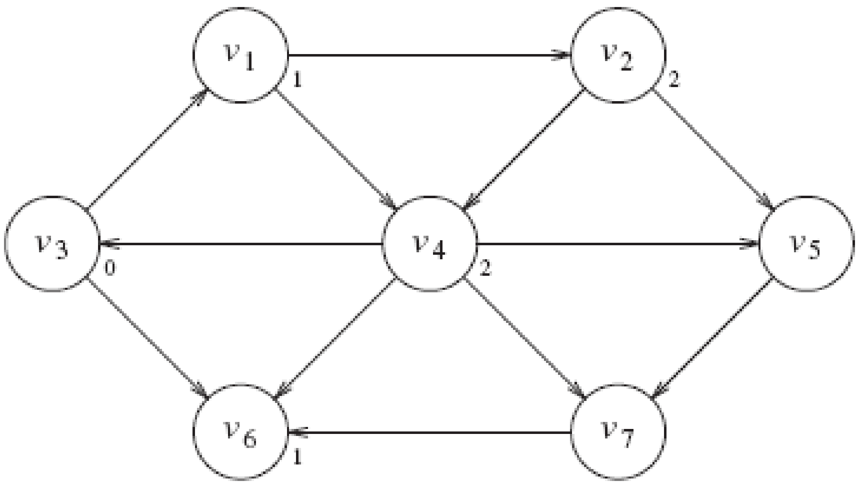


Figure 9.13 Graph after finding all vertices whose shortest path is 2

Unweighted Shortest Path, *cont'd*

- Find all vertices that are distance 3 from v_3.
  - Begin with the vertices adjacent to v_2 and v_4.

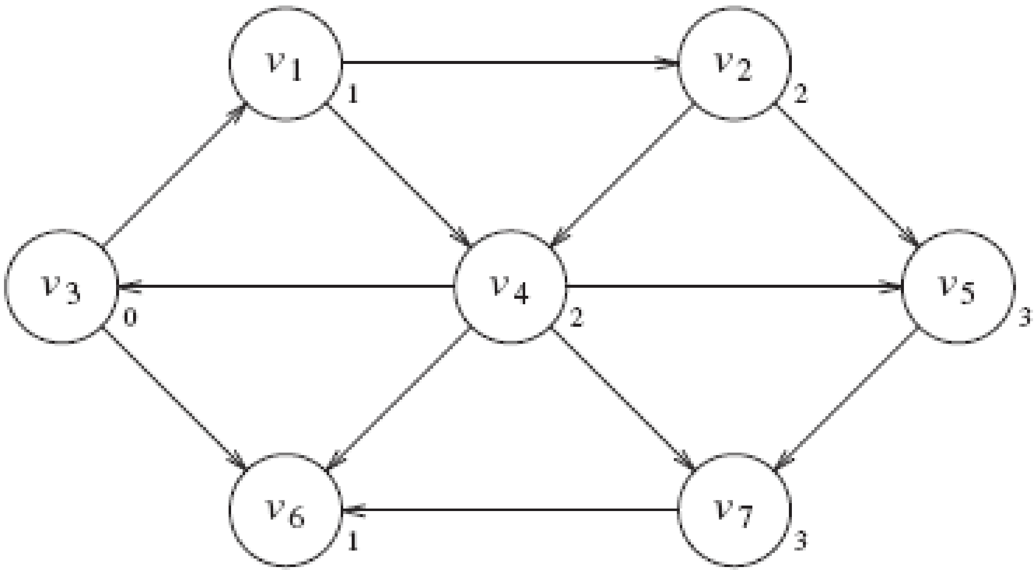


Figure 9.14 Final shortest paths

  - Now we have the <u>shortest paths</u> from v_3 to every other vertex.

Unweighted Shortest Path, *cont'd*

v	*known*	d_v	p_v
v_1	F	∞	0
v_2	F	∞	0
v_3	F	0	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

- Keep the tentative distance from vertex v_3 to another vertex in the d_v column.
- Keep track of the path in the p_v column.
- A vertex becomes known after it has been processed.
  - Don't reprocess a known vertex.
  - No cheaper path can be found.
- Set all $d_v = \infty$.
- Enqueue the distinquished vertex s and set $d_s = 0$.
- During each iteration, dequeue a vertex v.
  - Mark v as known.
  - For each vertex w adjacent to v whose $d_w = \infty$
    - Set its distance d_w to $d_v + 1$
    - Set its path p_w to v.
    - Enqueue w.

Data Structures and Algorithm Analysis in C++, 4th ed.
by Mark Allen Weiss
Pearson Education, Inc., 2014

Unweighted Shortest Path, *cont'd*

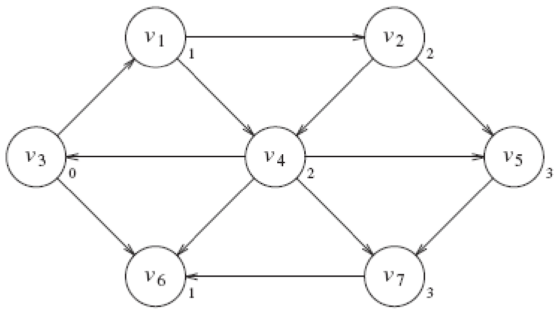


Figure 9.14 Final shortest paths

	Initial State			v_3 Dequeued			v_1 Dequeued			v_6 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	F	∞	0	F	1	v_3	T	1	v_3	T	1	v_3
v_2	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_3	F	0	0	T	0	0	T	0	0	T	0	0
v_4	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_5	F	∞	0	F	∞	0	F	∞	0	F	∞	0
v_6	F	∞	0	F	1	v_3	F	1	v_3	T	1	v_3
v_7	F	∞	0	F	∞	0	F	∞	0	F	∞	0
Q:	v_3			v_1, v_6			v_6, v_2, v_4			v_2, v_4		

	v_2 Dequeued			v_4 Dequeued			v_5 Dequeued			v_7 Dequeued		
v	known	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_2	T	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_3	T	0	0	T	0	0	T	0	0	T	0	0
v_4	F	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_5	F	3	v_2	F	3	v_2	T	3	v_2	T	3	v_2
v_6	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_7	F	∞	0	F	3	v_4	F	3	v_4	T	3	v_4
Q:	v_4, v_5			v_5, v_7			v_7			empty		

Figure 9.19 How the data change during the unweighted shortest-path algorithm

Data Structures and Algorithm Analysis in C++, 4th ed.
by Mark Allen Weiss
Pearson Education, Inc., 2014

Unweighted Shortest Path, *cont'd*

```
void unweighted( Vertex s )
{
    Queue<Vertex> q = new Queue<Vertex>( );

    for each Vertex v
        v.dist = INFINITY;

    s.dist = 0;
    q.enqueue( s );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );

        for each Vertex w adjacent to v
            if( w.dist == INFINITY )
            {
                w.dist = v.dist + 1;
                w.path = v;
                q.enqueue( w );
            }
    }
}
```

Figure 9.18 Pseudocode for unweighted shortest-path algorithm

Data Structures and Algorithm Analysis in C++, 4th ed.
by Mark Allen Weiss
Pearson Education, Inc., 2014

Break

Weighted Least Cost Path

- Dijkstra's Algorithm
  - Example of a greedy algorithm.
- Greedy algorithm
  - At each stage, do what appears to be the best at that stage.
  - A greedy algorithm may not always work.
- Keep the same information for each vertex:
  - Either known or unknown
  - Tentative distance d_v
  - Path information p_v

Dijkstra's Algorithm

- At each stage:
  - Select an unknown vertex *v* that has the smallest d_v.
  - Declare that the shortest path from *s* to *v* is known.
  - For each vertex *w* adjacent to *v*:
    - Set its distance d_w to the d_v + $\text{cost}_{v,w}$
    - Set its path p_w to *v*.

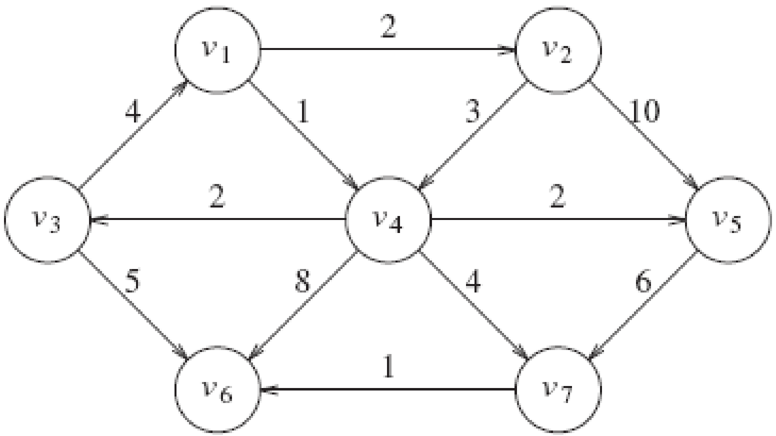


Figure 9.20 The directed graph *G* (again)

Dijkstra's Algorithm, *cont'd*

v	*known*	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

Figure 9.21 Initial configuration of table used in Dijkstra's algorithm

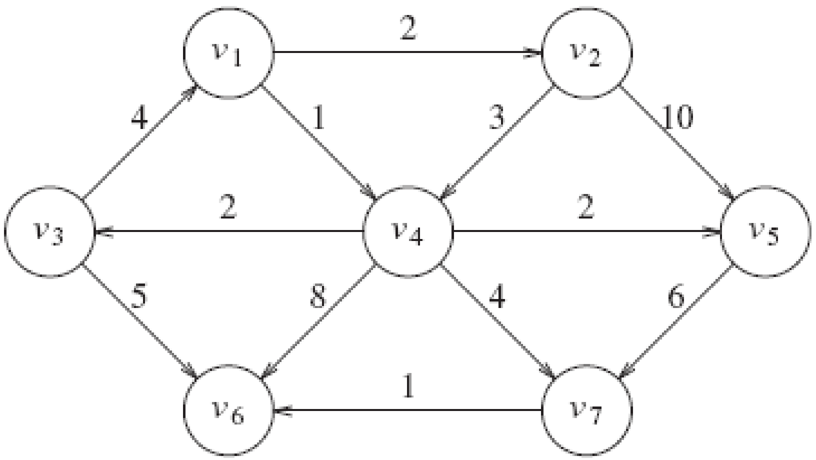


Figure 9.20 The directed graph G (again)

Start with $s = v_1$

Dijkstra's Algorithm, *cont'd*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	∞	0
v_4	F	1	v_1
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

Figure 9.22 After v_1 is declared *known*

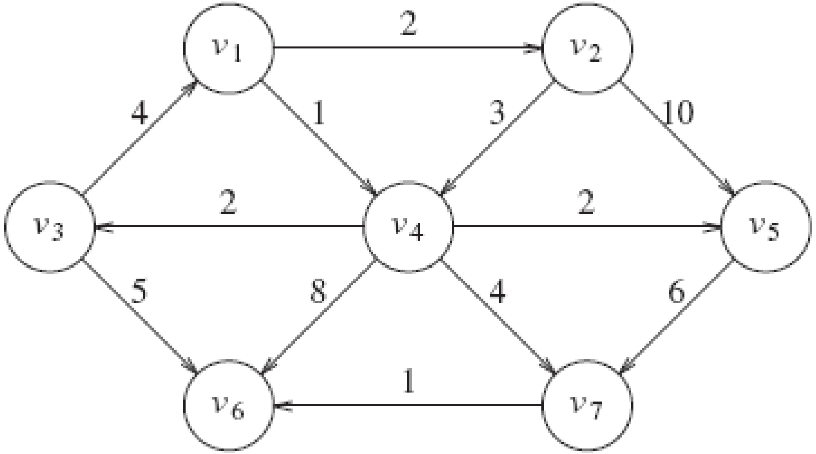


Figure 9.20 The directed graph G (again)

Set v_1 to known.
v_2 and v_4 are unknown and adjacent to v_1:
- Set d_2 and d_4 to their costs + cost of v_1
- Set p_2 and p_4 to v_1.

Dijkstra's Algorithm, *cont'd*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

Figure 9.23 After v_4 is declared *known*

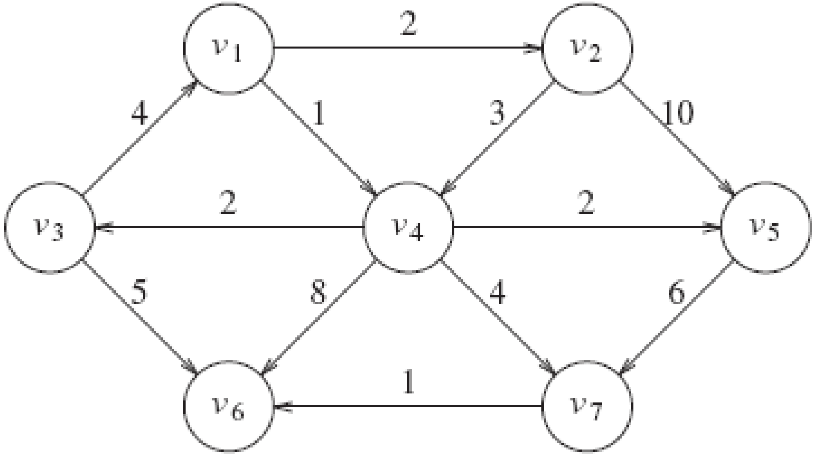


Figure 9.20 The directed graph G (again)

d_4 was the smallest unknown. Set v_4 to known.
v_3, v_5, v_6, and v_7 are unknown and adjacent to v_4:

- Set their d_w to their costs + cost of v_4
- Set their p_w to v_4.

Dijkstra's Algorithm, *cont'd*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

Figure 9.24 After v_2 is declared *known*

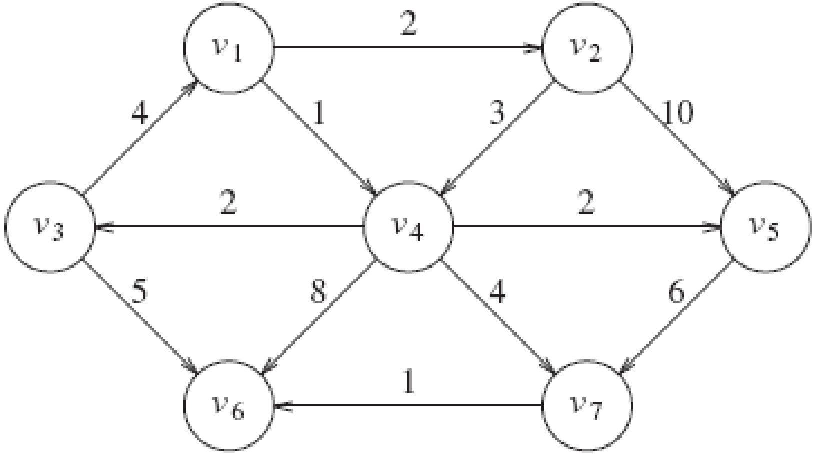


Figure 9.20 The directed graph G (again)

d_2 was the smallest unknown. Set v_2 to known.
v_5 is unknown and adjacent:

- d_5 is already 3 which is less than 2+10=12, so do not change v_5

Dijkstra's Algorithm, *cont'd*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	8	v_3
v_7	F	5	v_4

Figure 9.25 After v_5 and then v_3 are declared *known*

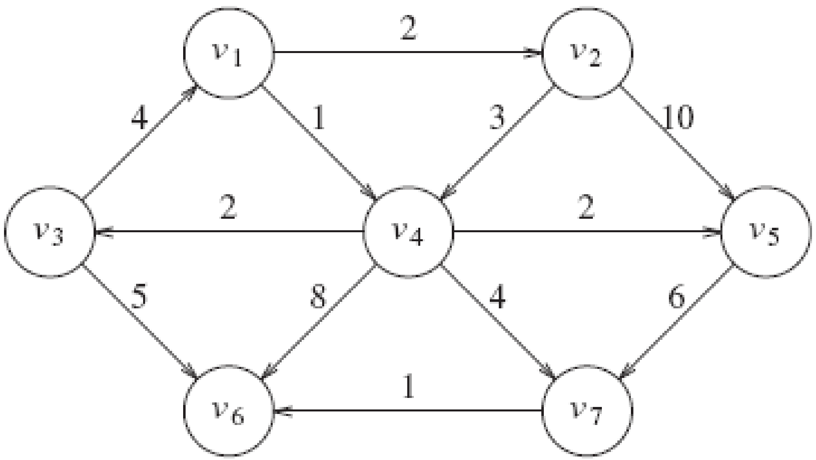


Figure 9.20 The directed graph G (again)

Set v_5 to known. v_7 is unknown and adjacent.
- Do not adjust since 5 < 3+6.

Set v_3 to known. v_6 is unknown and adjacent.
- Change d_6 to 3+5=8 which is less than its previous value of 9.
- Change p_6 to v_3.

Dijkstra's Algorithm, *cont'd*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	6	v_7
v_7	T	5	v_4

Figure 9.26 After v_7 is declared *known*

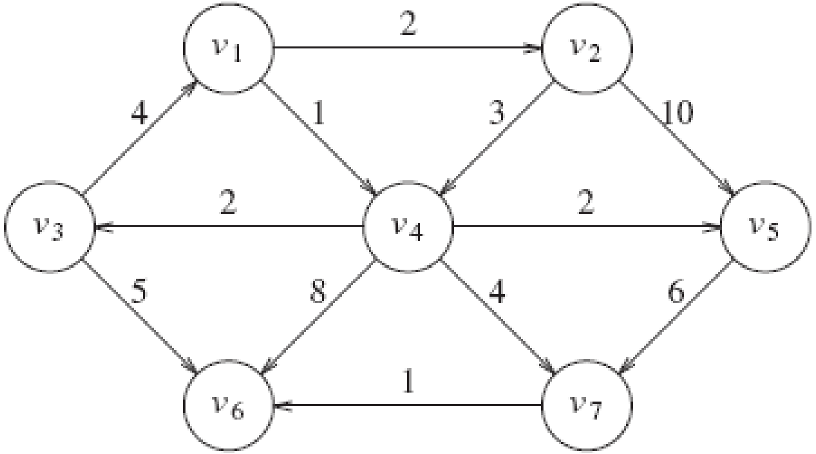


Figure 9.20 The directed graph G (again)

Set v_7 to known. v_6 is unknown and adjacent.

- Change d_6 to 5+1=6 which is less than its previous value of 8.
- Change p_6 to v_7.

Dijkstra's Algorithm, *cont'd*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	T	6	v_7
v_7	T	5	v_4

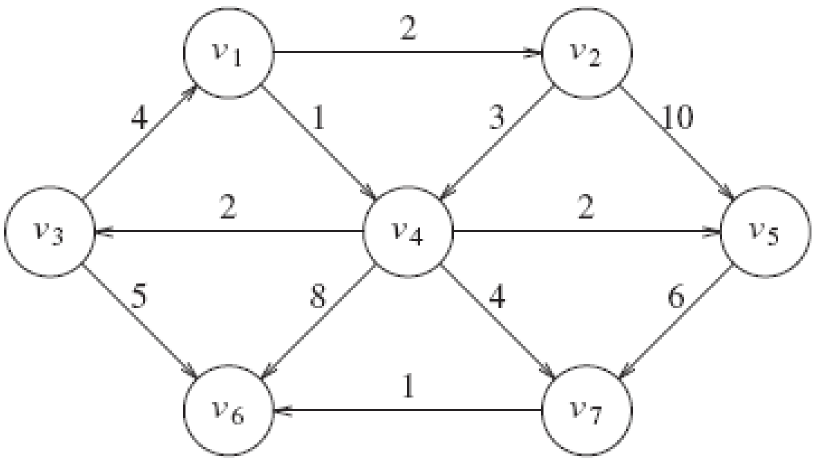


Figure 9.20 The directed graph G (again)

Set v_6 to known.
The algorithm terminates.

Figure 9.27 After v_6 is declared *known* and algorithm terminates

Dijkstra's Algorithm, *cont'd*

```
class Vertex
{
    public List     adj;      // Adjacency list
    public boolean  known;
    public DistType dist;     // DistType is probably int
    public Vertex   path;
        // Other fields and methods as needed
}
```

Figure 9.29 Vertex class for Dijkstra's algorithm

Dijkstra's Algorithm, *cont'd*

```
/*
 * Print shortest path to v after dijkstra has run.
 * Assume that the path exists.
 */
void printPath( Vertex v )
{
    if( v.path != null )
    {
        printPath( v.path );
        System.out.print( " to " );
    }
    System.out.print( v );
}
```

Figure 9.30 Routine to print the actual shortest path

Dijkstra's Algorithm, *cont'd*

```
void dijkstra( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    while( there is an unknown distance vertex )
    {
        Vertex v = smallest unknown distance vertex;

        v.known = true;

        for each Vertex w adjacent to v
            if( !w.known )
            {
                DistType cvw = cost of edge from v to w;

                if( v.dist + cvw < w.dist )
                {
                    // Update w
                    decrease( w.dist to v.dist + cvw );
                    w.path = v;
                }
            }
    }
}
```

Figure 9.31 Pseudocode for Dijkstra's algorithm

Minimum Spanning Tree (MST)

- Suppose you're wiring a new house.
  - What's the <u>minimum length of wire</u> you need to purchase?
- Represent the house as an undirected graph.
  - Each electrical outlet is a vertex.
  - The wires between the outlets are the edges.
  - The cost of each edge is the length of the wire.

Minimum Spanning Tree (MST), *cont'd*

- Create a tree formed from the edges of an undirected graph that connects all the vertices at the lowest total cost.

Minimum Spanning Tree (MST), *cont'd*

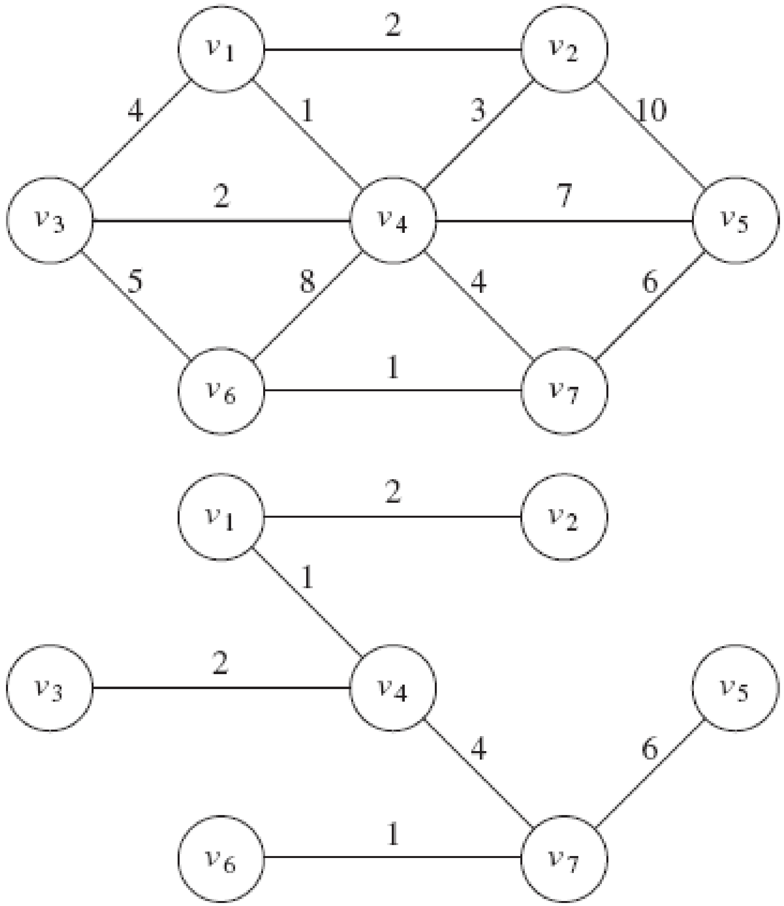


Figure 9.50 A graph G and its minimum spanning tree

- The MST
  - Is an acyclic tree.
  - Spans (includes) every vertex.
  - Has $|V|$-1 edges.
  - Has <u>minimum total cost</u>.

Minimum Spanning Tree (MST), *cont'd*

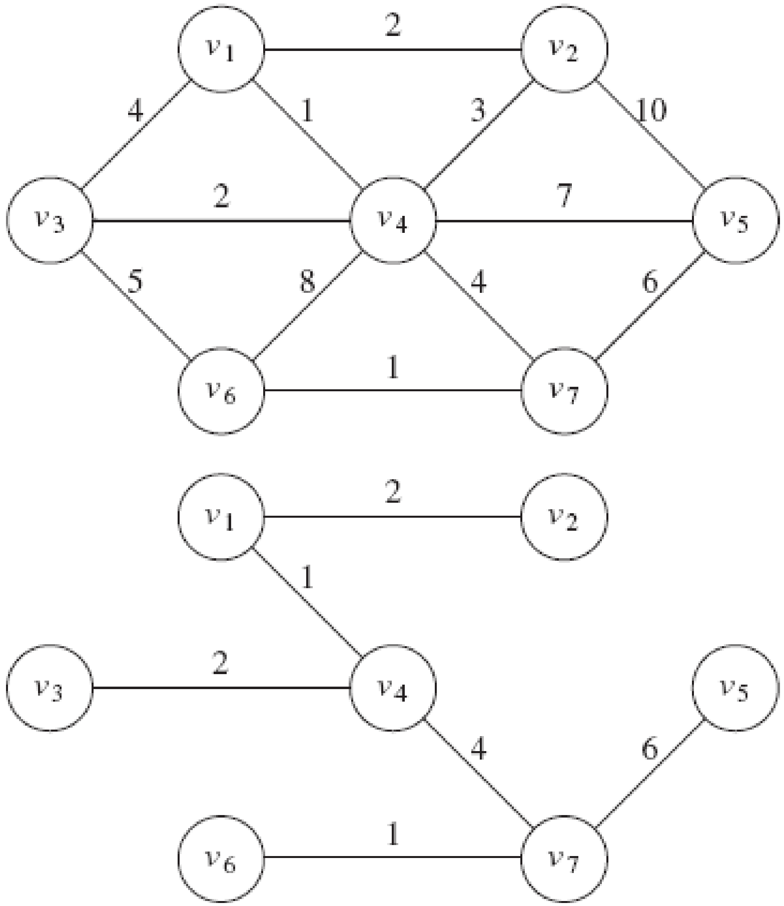


Figure 9.50 A graph G and its minimum spanning tree

- Add each edge to an MST in such a way that:
  - It does not create a cycle.
  - Is the least cost addition.
- A <u>greedy</u> algorithm!

Prim's Algorithm for MST

- Rediscovered by Robert C. Prim in 1957 to solve connection network problems.
  - First discovered in 1930 by Czech mathematician Vojtěch Jarník.
- At any point during the algorithm, some vertices are in the MST and others are not.
- Choose one vertex to start.

Prim's Algorithm for MST, *cont'd*

- At each stage, add another vertex to the tree.
- Choose a vertex such that:
  - The edge (*u*, *v*) has the lowest cost among all the edges.
  - *u* is already in the tree and *v* is not.
- Similar to Dijkstra's algorithm for shortest paths.
  - Maintain whether or not a vertex is known, and its d_v and p_v values.

Prim's Algorithm for MST, *cont'd*

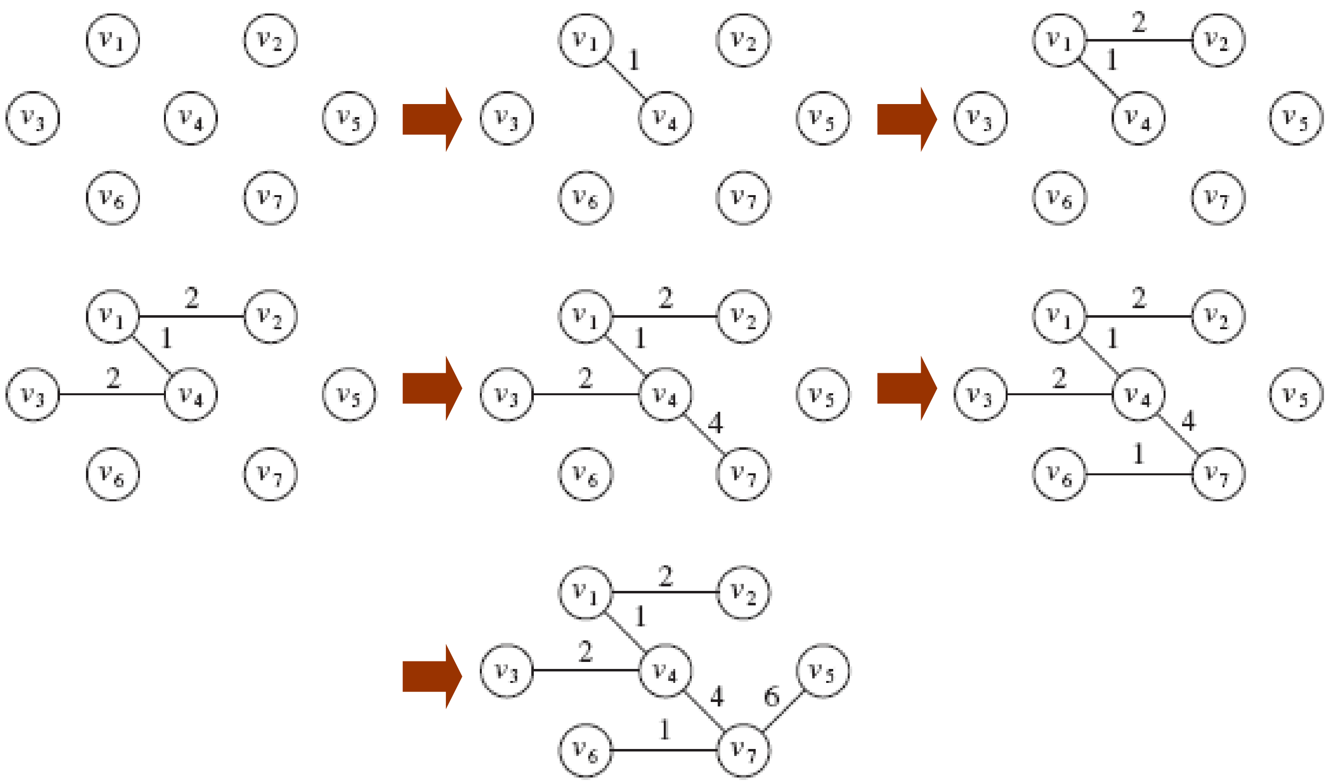


Figure 9.51 Prim's algorithm after each stage

Data Structures and Algorithm Analysis in C++, 4th ed.
by Mark Allen Weiss
Pearson Education, Inc., 2014

Prim's Algorithm for MST, *cont'd*

v	*known*	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

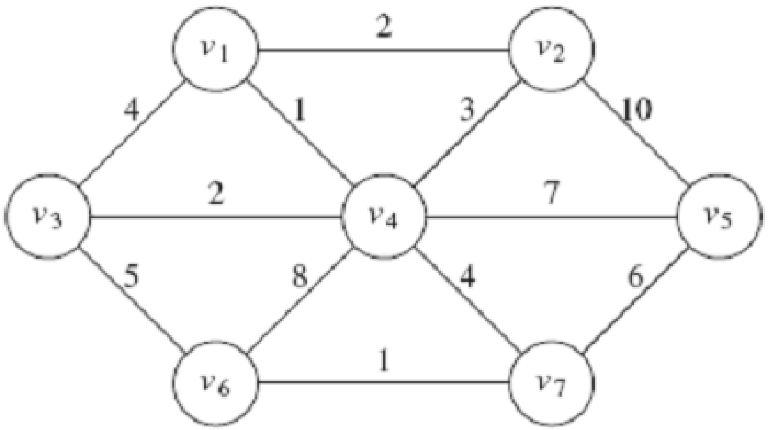
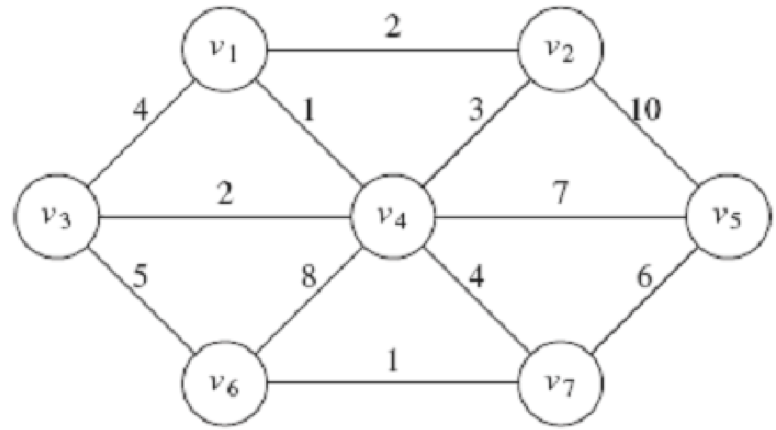


Figure 9.52 Initial configuration of table used in Prim's algorithm

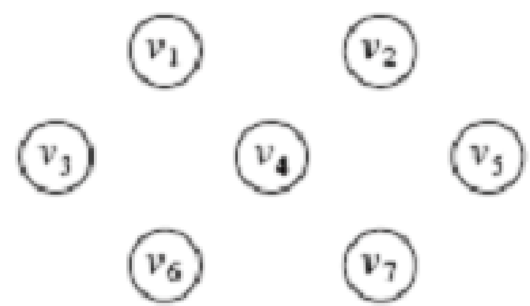
Prim's Algorithm for MST, *cont'd*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	4	v_1
v_4	F	1	v_1
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

Figure 9.53 The table after v_1 is declared *known*



Choose v_1 to start. Declare it known.
Set the d_v and p_v of v_1's neighbors.



Prim's Algorithm for MST, *cont'd*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	2	v_4
v_4	T	1	v_1
v_5	F	7	v_4
v_6	F	8	v_4
v_7	F	4	v_4

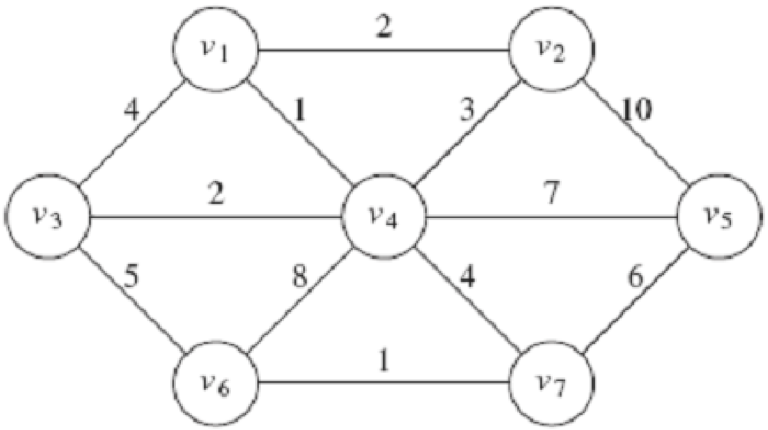


Figure 9.54 The table after v_4 is declared *known*

Choose v_4 and declare it known.
Set the d_v and p_v of v_4's neighbors that are still unknown: v_3, v_5, v_6, and v_7.
Don't do v_2 because $d_2 = 2 < 3$.

Prim's Algorithm for MST, *cont'd*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	F	7	v_4
v_6	F	5	v_3
v_7	F	4	v_4

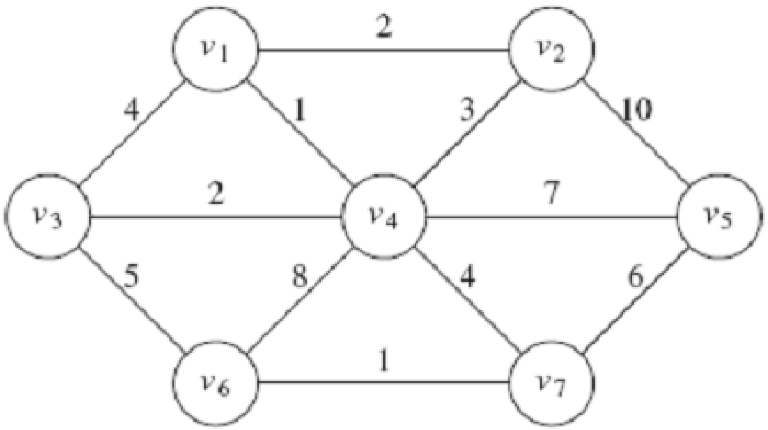


Figure 9.55 The table after v_2 and then v_3 are declared *known*

Choose v_2 and declare it known.
No changes to the table.

Choose v_3 and declare it known.
Set the d_v and p_v of v_3's neighbors that still unknown: v_6.
Set $d_6 = 5 <$ its previous value 8.

Prim's Algorithm for MST, *cont'd*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	F	6	v_7
v_6	F	1	v_7
v_7	T	4	v_4

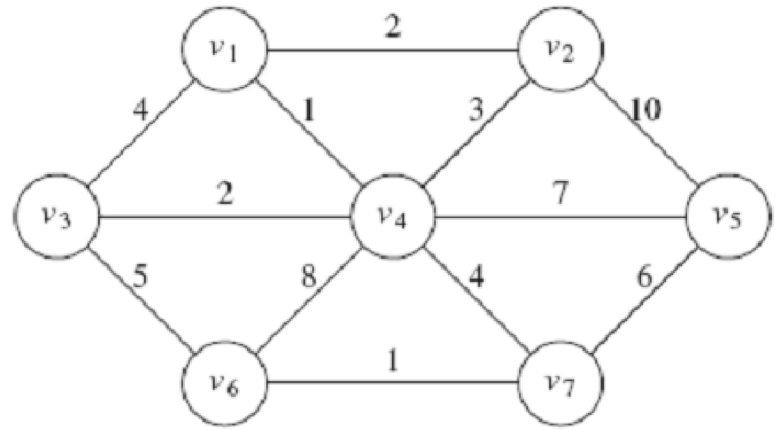


Figure 9.56 The table after v_7 is declared *known*

Choose v_7 and declare it known.
Set the d_v and p_v of v_4' s neighbors that still unknown: v_5 and v_6.
Set $d_5 = 6 <$ its previous value 7.
Set $d_6 = 1 <$ its previous value 5.

Prim's Algorithm for MST, *cont'd*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	T	6	v_7
v_6	T	1	v_7
v_7	T	4	v_4

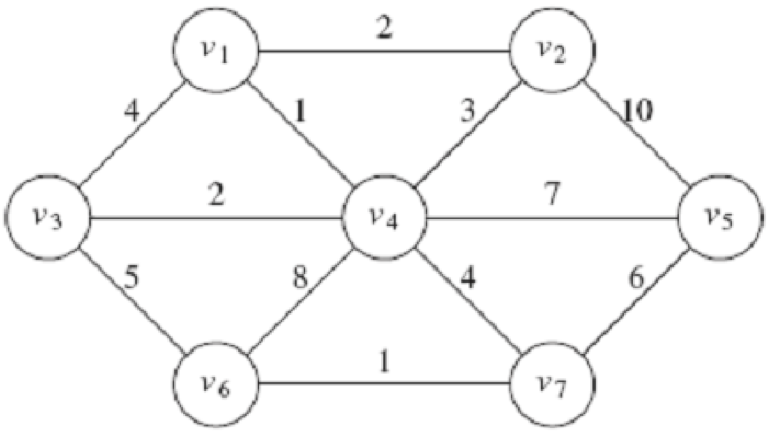
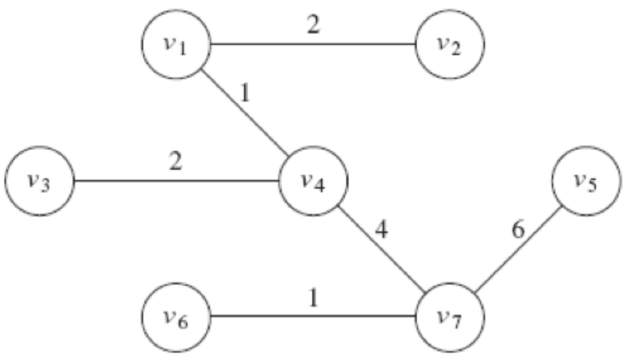


Figure 9.57 The table after v_6 and v_5 are selected (Prim's algorithm terminates)

Choose v_6 and declare it known.
No changes to the table.

Choose v_5 and declare it known.
No changes to the table.



Data Structures and Algorithm Analysis in C++, 4th ed.
by Mark Allen Weiss
Pearson Education, Inc., 2014

Graph Traversal Algorithms

- Graph traversal is similar to tree traversal.
  - Visit each vertex of a graph in a particular order.
- Special problems for graphs:
  - It may not be possible to reach all vertices from the start vertex.
  - The graph may contain cycles.
    - Don’t go into an infinite loop.
    - “Mark” each vertex after a visit.
    - Don’t revisit marked vertices.

You’re Lost in a Maze

- You have a bag of bread crumbs.
- As you go down each path,
  you drop bread crumbs to <u>mark your path</u>.
- Whenever you come to a dead end, you retrace your path by following your bread crumbs.
- You continue retracing your path (“<u>backtracking</u>”) until you come to an intersection with an unmarked path.
- You (<u>recursively</u>) go down the unmarked path.

Depth-First Search

- Represent the maze as a graph.
  - Each path is an edge.
  - Each intersection is a vertex.
- You are doing a depth-first search of the graph.

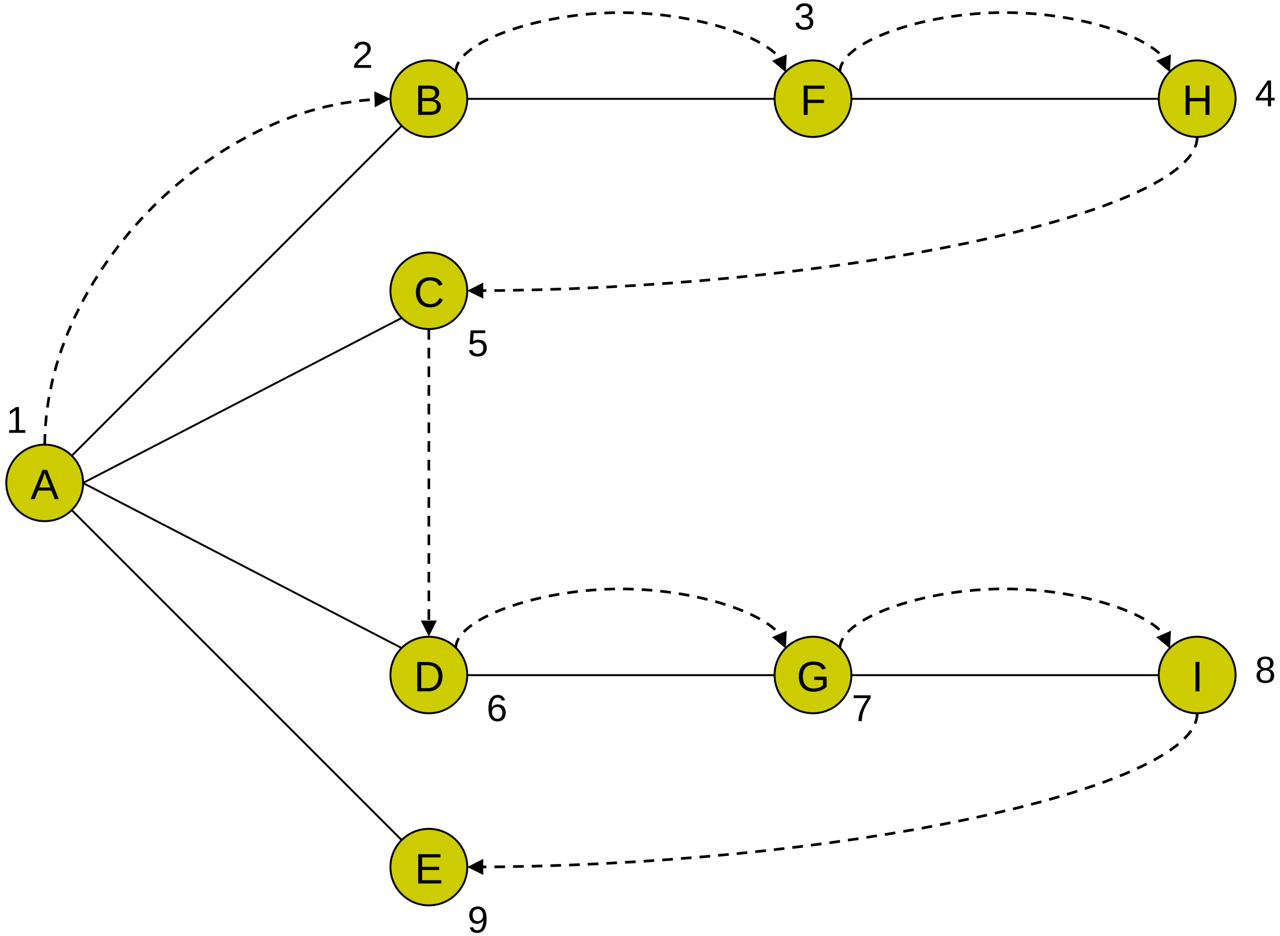
Depth-First Search

```
void dfs(Vertex v)
{
    v.visited = true;  // mark

    for each Vertex w adjacent to v {
        if (!w.visited) {
            dfs(w);  // recursively visit w
        }
    }
}
```

- Visits each vertex once.
- Processes each edge once in a directed graph.
- Processes each edge from both directions in an undirected graph.
- Therefore, $O(|V| + |E|)$.

Depth-First Search



Depth-First Search and Games

- Depth-first search is used by <u>game-playing</u> programs.
  - Example: IBM's "Deep Blue" chess playing program.
- Use a graph to represent the possible moves from the present situation into the future.
- Each vertex is a <u>decision point</u> for either you or your opponent.

Depth-First Search and Games, *cont'd*

- Perform a depth-first search to look at possible move outcomes of both you and your opponent.
- Each edge would have the cost of going down that path.
- Backtrack if a path is a dead end or its cost is not beneficial.
- How deeply your program can search depends on the computer's memory and the allowed search time.

Find a Lost Child in a Large Building

- Start in the room where the child was last seen.
- Search each room adjacent to the first room.
  - Put a tag on the door to mark a room you've already searched.
- Then search each room adjacent to the rooms you've already searched.
- Repeatedly search all the rooms adjacent to rooms you've already searched before moving farther out from the first room.

Breadth-First Search

- Represent the building as a graph.
  - Each room is a vertex.
  - Each hallway between rooms is an edge.
- You are doing a breadth-first search of the graph.

Breadth-First Search

```
void bfs(Vertex s)
{
    Queue<Vertex> q = new Queue<>();
    q.enqueue(s);
    s.visited = true;

    while (!q.empty()) {
        Vertex v = q.dequeue();

        for each Vertex w adjacent to v {
            if (!w.visited) {
                w.visited = true;
                q.enqueue(w);
            }
        }
    }
}
```

Breadth-First Search