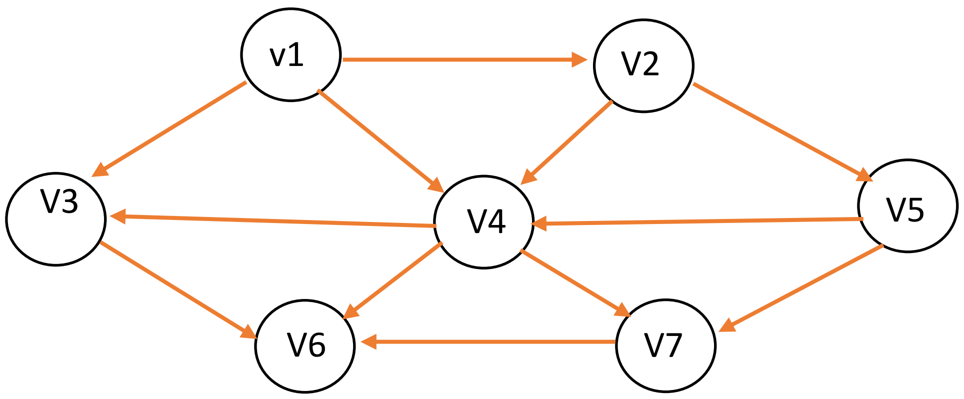
Chapter 5
Graph Algorithm

Definitions

- **A graph G = (V, E)** consists of a set of vertices, V, and a set of edges, E.



- Each edge is a pair (v , w), where v, w ϵ V. E edges are sometimes referred to as arcs.
- If the pair is ordered, then the graph is **directed.** Directed graphs are sometimes referred to as **diagraphs**.

- Vertex w is **adjacent** to v, if and only if (v, w) ϵ E.
- In an undirected graph with edge (v, w), and hence (w, v), w is adjacent to v and v is adjacent to w.

- Sometimes an edge has a third components kwon as either a **weight** or a **cost**.

- A **path** in a graph is a sequence of vertices w_1, w_2, w_3,…, w_n such that (w_i, w_{i+1}) ϵ E for 1 <= i <= n.
- The **length** of such a path is the number of edged on the path, which is equal to n-1.

- If the graph contains an edge (v, v) from a vertex to itself, then the path v, v is sometimes referred to as a **loop**.

- A **simple path** is a path such that all vertices are distinct, except that the first and last could be the same.

- A **cycle** in a directed graph is a path of length at least 1, such that w_1 = w_n, this cycle is simple if the path is simple.

- A directed graph is **acyclic** if it has no cycles. A directed acyclic graph is sometimes referred to by its abbreviation, **DAG**.

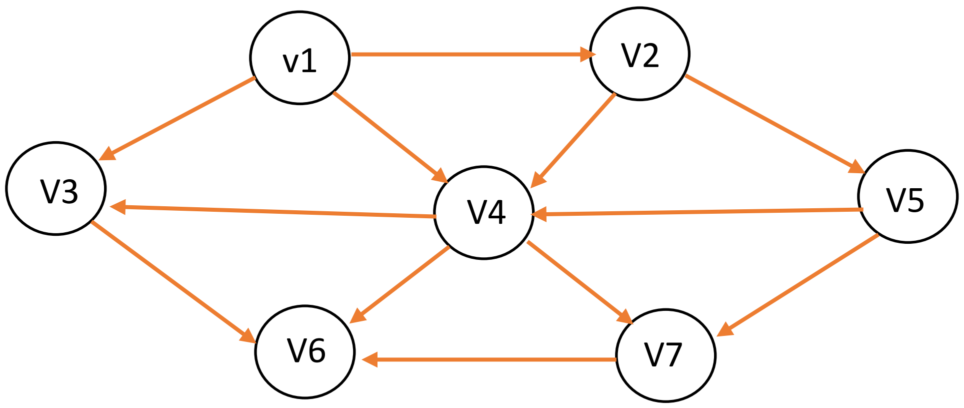
- An undirected graph is **connected** if there is a path from every vertex to every other vertex.
- A directed graph with this property is called **strongly connected**.
- If the directed graph is not strongly connected, but the underlying graph (without direction to the arcs) is connected, then the graph is said to be **weakly connected**.

- A **complete graph** is a graph in which there is an edge between every pair of vertices.

Examples:

- Airport system can be modeled by a graph.
- Traffic flow can be modeled by a graph.

Representation of graphs



1. Adjacency matrix representation
    1. One simple way to represent a graph is to use a two dimensional array.

    2. For each edge (u, v), we set a[u][v] = 1; otherwise the entry in the array is 0.

    3. If the edge has a weight associated with it, then we can set a[u][v] = to the weight and use either a very large or a very small weight as a sentinel to indicate non exit tent edge.

4. The space requirement is O($|V|^2$), which can be prohibitive if the graph does not have very many edges.

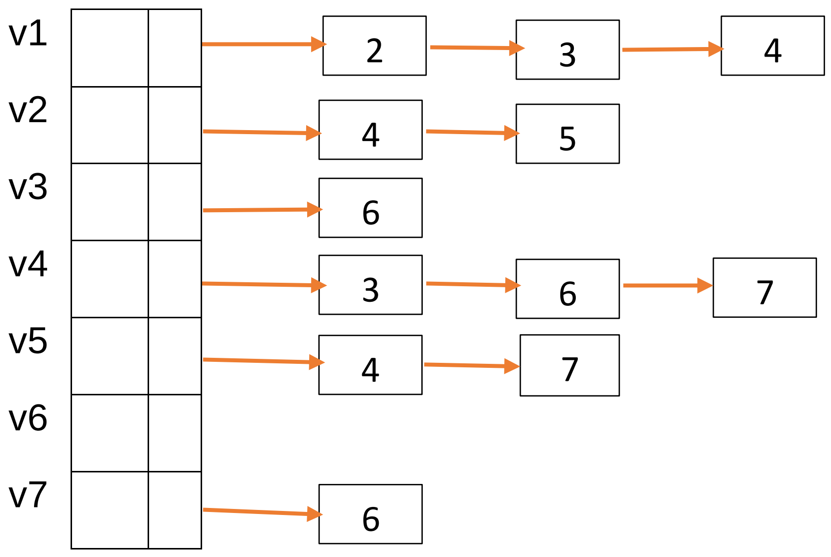
5. An adjacency matrix is an appropriate representation if the graph is dense ➔ $|E| = O(|V|^2)$.

6. In most of the applications that we shell see, this is not true. For instance, suppose the graph represents a street map. Where almost all the streets run either north-south or east-west. Therefore, any intersection is attached to roughly four streets, so if the graph is directed and all streets are two-way, then $|E| \approx 4|V|$. If there are 3,000 intersections, then we have a 3,000 vertex graph with 12,000 edges entries, which would require an array of size 9,000,000.

	v1	v2	v3	v4	v5	v6	v7
v1	0	1	1	1	0	0	0
v2	0	0	0	1	1	0	0
v3	0	0	0	0	0	1	0
v4	0	0	1	0	0	1	1
v5	0	0	0	1	0	0	1
v6	0	0	0	0	0	0	0
v7	0	0	0	0	0	1	0

2. Adjacency list representation

- If the graph is not dense, in other words, if the graph is **sparse**.
- For each vertex, we keep a list of all adjacent vertices.
- The space requirement is then (|E| + |V|).
- If the edges have weights, then this additional information is also stored in the cells.
- Adjacency lists are the standard way to represent graphs.
- In most real life applications, the vertices have names, which are unknown at compile time. Since we cannot index an array by a unknown names, we must provide a mapping of names to numbers. The easiest way to do this is to use hash table.

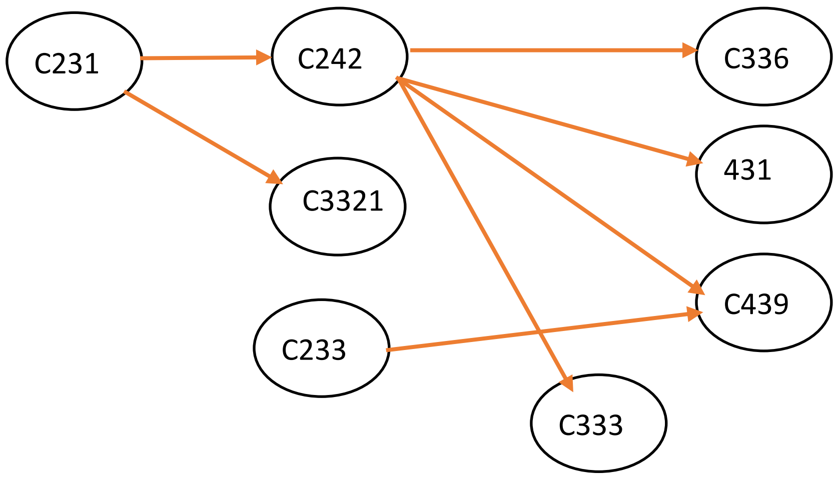


Topological Sort

A topological sort is an ordering of vertices in a directed acyclic graph, such that if there is a path from vi to v_j, then v_j appears after vi in the ordering.

Example:

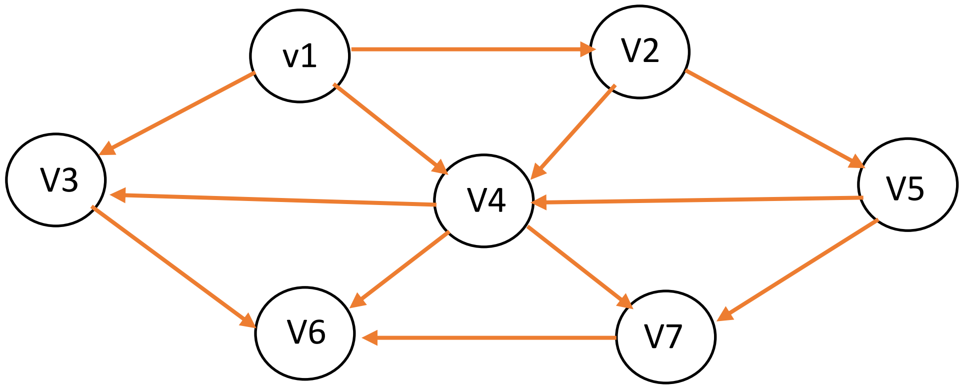
Represents the course prerequisite structure at the Birzeit university:



- It is clear that a topological ordering is not possible if the graph has a cycle, since for two vertices v and w on the cycle, v precedes w and w precedes v.

- The ordering is not necessarily unique; any legal ordering will do.

An acyclic graph



Order: v1 -> v2 -> v5 -> v4 -> v3 -> v7 -> v6

And

V1 -> v2 -> v5 -> v4 -> v7 -> v3 -> v6

Are both topological orderings

- A simple algorithm to find a topological ordering is first to find any vertex with no incoming edges. We can then print this vertex, and remove it, along with its edges, from the graph. Then we apply this same strategy to the rest of the graph.

- To formalize this, we define the indegree of a vertex v as the number of edges (u, v). We compute the indegrees of all vertices in the graph. Assume that the indegree array is initialized and that the graph is read into an adjacency list.

Algorithm

```
void topSort (Graph G)
    int counter;
    Vertex v, w;
    for ( counter = 0; counter <= numOfVertex; counter++)
            v = findMinVertexOfIndegreeZero();
            if ( v == notAVertex )
                    error (“ Graph has a cycle “);
                    break;
            end if
            topNum [v] = counter;
            for each w adjacent to v
                    indegree[ w ] --;
            end for
    end for
end.
```

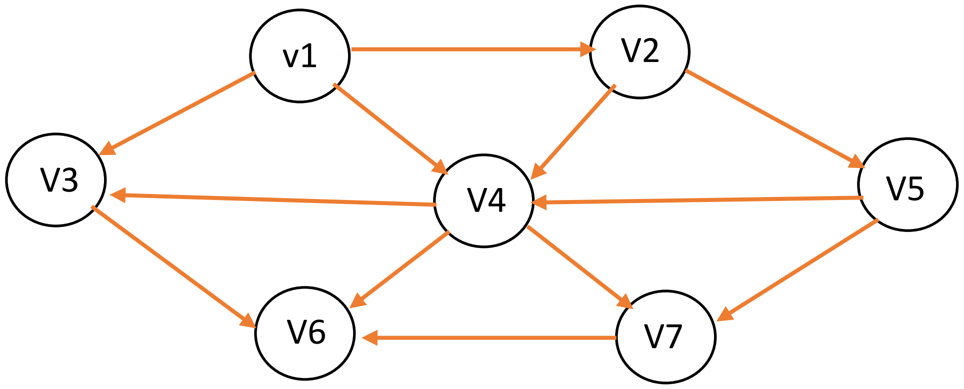
- Because findMinVertexOfIndegreeZero() is a simple sequential scan of the indegree array, each call to it takes O(|V|) time. Since there are |V| such calls, the running time of the algorithm is O($|V|^2$).

Algorithm 2

- The time to perform this algorithm is O(|E| + |V|) if adjacency lists are used. This is apparent where one realizes that the body of the for loop is executed at most once per edge. The queue operations are done at most once per vertex, and the initialization steps also take time proportional to the size of the graph

```
void topSort ( Graph G)
    Queue Q;
    Int counter;
    Vertex v, w;
    Q = createQueue( numVertex);
    makeNull(Q);
    counter = 0;
    for each vertex v
        if ( indegree[ v ] == 0 )
            enqueue(v, Q);
        end if
    end for
    while ( Not isEmpty( Q ))
        v = dequeue( Q );
        topNum[ v ] = ++counter;
        for each w adjacent to v
            if ( --indegree[ w ] == 0)
                enqueue(w,Q);
            end if
        end for
    end while
    if ( counter != numVertex )
        error( “ Graph has a cycle”);
    end if
    diposeQueue( Q );
end.
```

Example

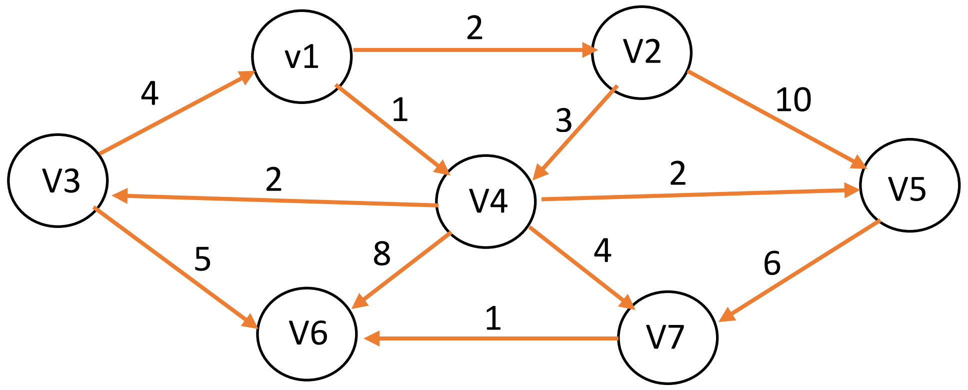


Vertex	1	2	3	4	5	6	7
v1	0	0	0	0	0	0	0
v2	1	0	0	0	0	0	0
v3	2	1	1	1	0	0	0
v4	3	2	1	0	0	0	0
v5	1	1	0	0	0	0	0
v6	3	3	3	3	2	1	0
v7	2	2	2	1	0	0	0
enqueue	v1	v2	v5	v4	v3, v7		v6
dequeue	v1	v2	v5	v4	v3	v7	v6

Shortest Path Algorithm

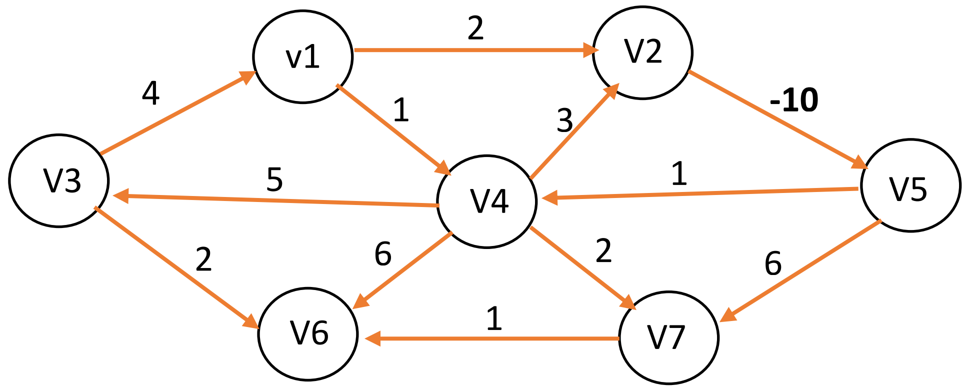
- The cost of a path $v_1, v_2, \dots, v_n$ is $\sum_{i=1}^{n-1} c_i, c_{i+1}$. This is referred to as the weighted path length.
- The unweighted path length is merely the number of edges on the path, (n-1).

For example, the shortest weighted path from v1 to v6 has a cost of 6 and goes from v1 to v4 to v7 to v6. The shortest unweighted path between these vertices is 2.



- The graph below shoes the problems that negative edges can cause. The path from v5 to v4 has cost 1, but a shorter path exists by following the loop v5, v4, v2, v5, v4, which has cost -5. This path is still not the shortest, because we could stay in the loop arbitrarily long. Thus, the shortest path between these two points is undefined.

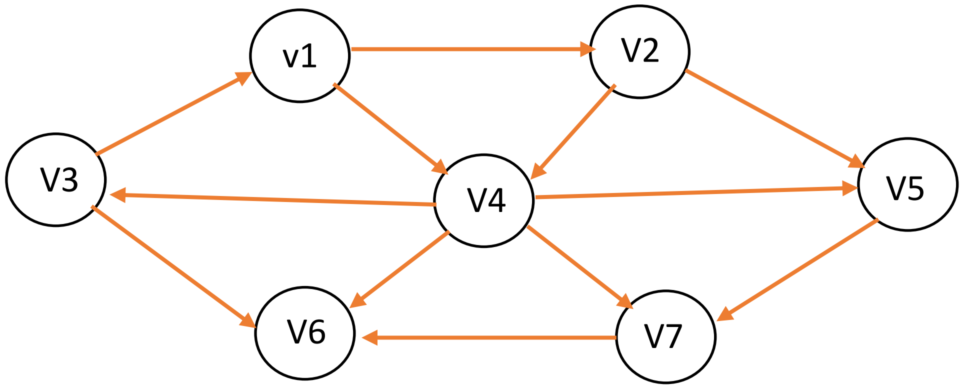
- This loop is known as a negative cost cycle; When one is present in the graph, the shortest path are not defined.



- We will examine algorithms to solve four versions of this problem.
  - First, we will consider the unweighted shortest path problem and show how to it in O (|E| + |V|).
  - Next, we will show how to solve the weighted shortest path problem if we assume that there are no negative edges. The running time for this algorithm is O (|E| log|V|) when implemented with reasonable data structures.
  - If the graph has negative edges, we will provide a simple solution, which unfortunately has a poor time bound of O (|E|. |V|).
  - Finally, we will solve the weighted problem for the special case of acyclic graphs in the linear time.

1) Unweighted shortest paths

We would like to find the shortest path from **s** to all other vertices. We are only interested in the number of edges contained on the path. So there are no weights on the edges. This is clearly a special case of the weighted shortest path problem, since we could assign all edges a weight of 1.



Initial configuration of table used in unweighted shortest path computation

V	Known	dv	pv
v1	0	∞	0
v2	0	∞	0
v3	0	0	0
v4	0	∞	0
v5	0	∞	0
v6	0	∞	0
v7	0	∞	0

Suppose we choose **s** to be v3

Algorithm #1

```
void unweighted ( Table T)
    int currentDist;
    Vertex V, W;
    for ( currentDist = 0; currentDist < numVertex; currentDist++)
      for each vertex v
          if ( ( !T[ v ].known ) And ( T[ v ].dist == currentDist )
               T[ v ].known = true;
               for each w adjacent to v
                    if ( T[ w ].dist == IntMax )
                         T[ w ].dist = currentDist + 1;
                         T[ w ].path = v;
                    end if
               end for
          end if
      end for
    end for
end
```

- The running time of the algorithm is $O(V^2)$, because of the doubly nested for loop.

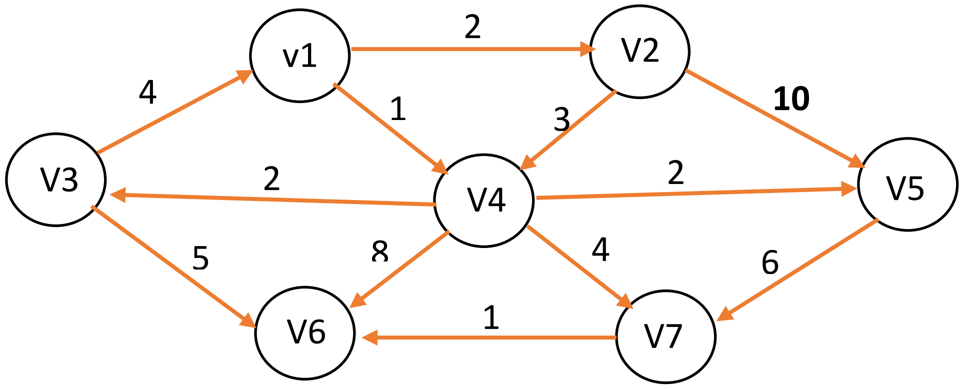
Algorithm #2

```
void unweighted ( Table T)
    Queue Q;
    Vertex v, w;
    Q = createQueue ( numVertex);
    makeNull (Q);
    // enqueue the start vertex s,
    enqueue(Q, s);
    while ( ! isEmpty (Q))
        v = dequeue (Q);
        T[ v ].known = true;
        for each w adjacent to v
            if ( T[ w ].dist == IntMax )
                T[ w ].dist = T[ v ].dist +1;
                T [w ].path = v;
                enqueue (Q, w);
            end if
        end for
    end while
    dispaoseQueue (Q);
end.
```

- The running time of the algorithm is O (|E| + |V|)

2) Dijkstra’s Algorithm

- If the graph is weighted, the problem becomes harder, but we can still use the ideas from the unweighted case.
- This solution is a example of a greedy algorithms generally solve a problem in stages by doing what appears to be the best thing at each stage.



V	known	d_v	p_v	known	d_v	p_v	known	d_v	p_v	known	d_v	p_v	known	d_v	p_v	known	d_v	p_v	known	d_v	p_v
v1	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
v2	0	∞	0	0	2	v_1	0	2	v_1	1	2	v_1	1	2	v_1	1	2	v_1	1	2	v_1
v3	0	∞	0	0	∞	0	0	3	v_4	0	3	v_4	1	3	v_4	1	3	v_4	1	3	v_4
v4	0	∞	0	0	1	v_1	1	1	v_1	1	1	v_1	1	1	v_1	1	1	v_1	1	1	v_1
v5	0	∞	0	0	∞	0	0	3	v_4	0	3	v_4	1	3	v_4	1	3	v_4	1	3	v_4
v6	0	∞	0	0	∞	0	0	9	v_4	0	9	v_4	0	9	v_7	0	6	v_7	1	6	v_7
v7	0	∞	0	0	∞	0	0	5	v_4	0	5	v_4	0	5	v_4	1	5	v_4	1	5	v_4

Declarations for Dijkstra's Algorithm

```
int vertex;
tableEntry
{
        List header;
        boolean known;
        distType    dist;
        vertex path;
}
tableEntry  Table[ numberOfVertex + 1];

void intializeTable ( vertex start, Graph g, Table T)
begin
        int i;
        readGraph(G, T);
        for ( i = numberOfVertex; i > 0; i--)
                T[ i ].known = false;
                T[ i ].dist = INT_MAX;
                T[ i ].path = notVertex;
        end for
        T[ start ].dist = 0;
end.
```

```
void Dijkstra ( Table T )
begin
    vertex v, w;
    for ( ; ; )
        v = smallest_Unknown_Distance_Vertex;
        if ( v == notVertex )
            break;
        end if
        T[ v ].known = true;
        for each w adjacent to v
            if ( ! T[ w ].known )
                if ( T[ v ].dist + $c_{v,w}$ < T[ w ].dist )
                    T[ w ].dist = T[ v ].dist + $c_{v,w}$ ;
                    T[ w ].path = v;
                end if
            end if
        end for
    end for
end.
```

Time = $O(V^2)$

```
void printPath( vertex v, Table T )
    if ( T[ v ].path != notVertex )
        printPath( T[ v ].path, T);
        print( " to ");
    end if
    print( v );
end.
```

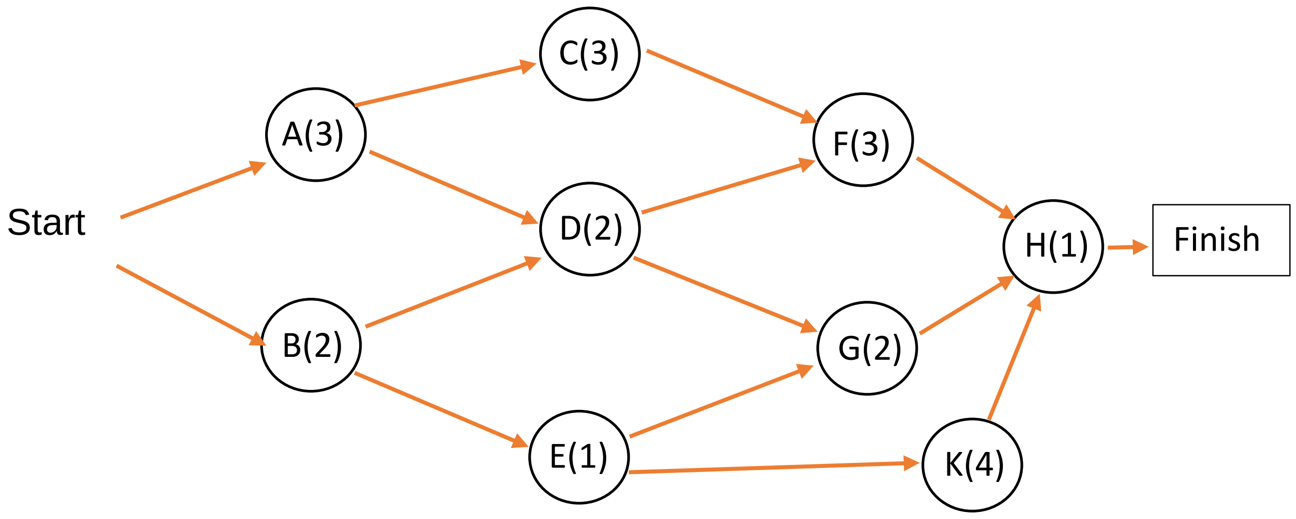
3) Graphs with Negative Edge costs

- If the graph has negative edge costs. Then Dijkstra's Algorithm does not work.
- The problem is that once a vertex **u** is declared known, it is possible that from some other unknown vertex **v** there is a path back to **u** that is very negative. In such a case, taking a path from **s** to **v** back to u is better than going from **s** to **u** without using **v**.
- A combination of the weighted and unweighted algorithms will solve the problem, but at the cost of a drastic increase in running time.
- We forget about the concept of known vertices, since our algorithm needs to be able to change its mind.
- The running time is O(E.V) if adjacency lists are used.

```
void weightesNegative( Table T)
    Queue Q;
    veretex v, w;
    Q = createQueue( numOfVertex );
    enqueue( Q, s);
    while ( ! isEmpty( Q ))
        v = dequeue( Q );
        for each w adjacent to v
            if( T[ v ].dist + $c_{v,w}$ < T[ w ].dist )
                T[ w ].dist = T[ v ].dist + $c_{v,w}$ ;
                T[ w ].path = v;
                if ( w is not already in Q )
                    enqueue( Q, w);
                end if
            end if
        end for
    end while
    disposeQueue( Q );
end.
```

4) Acyclic Graphs

- If the graph is known to be acyclic, we can improve Dijkstra's Algorithm by changing the order in which vertices are declared known, otherwise known as vertex selection rule.
- The new rule is to select vertices in topological order.
- The selection rule works because when a vertex **v** is selected, its distance, $\mathbf{d_v}$, can no longer be lowered, since by the topological ordering rule it has no incoming edge emarating from unknown nodes.
- A more important use of acyclic graphs is **critical path analysis**.

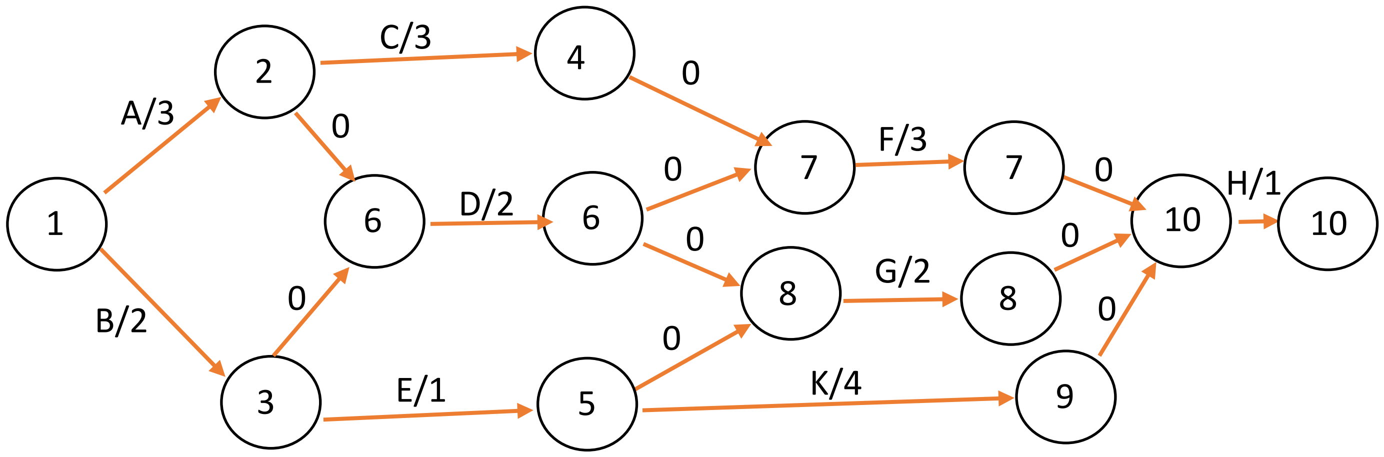


- This graph is thus known as an **activity node graph**.
- The edges represent procedure relationship.
- An edge (v, w) means that activity **v** must be completed before activity **w** may begin.

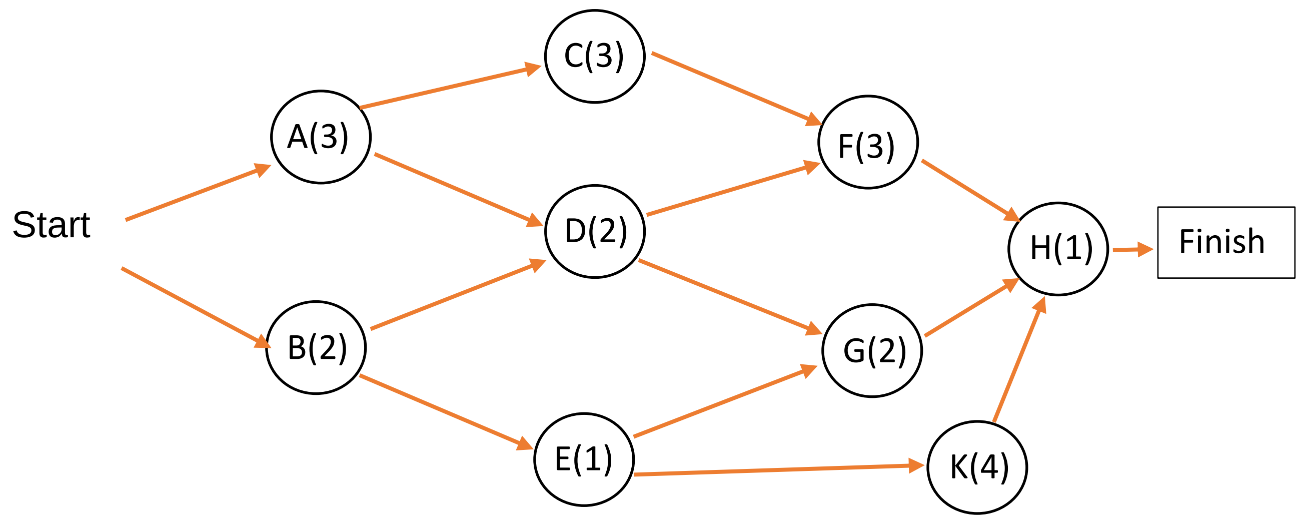
- There is several important questions which would be of interest to answer

    1. What is the earliest completion time for the project?
       We can see from the graph that 10 time units are required along the path A, C, F, H.
    2. Determine which activities can be delayed, and by how long without affecting the minimum completion time.

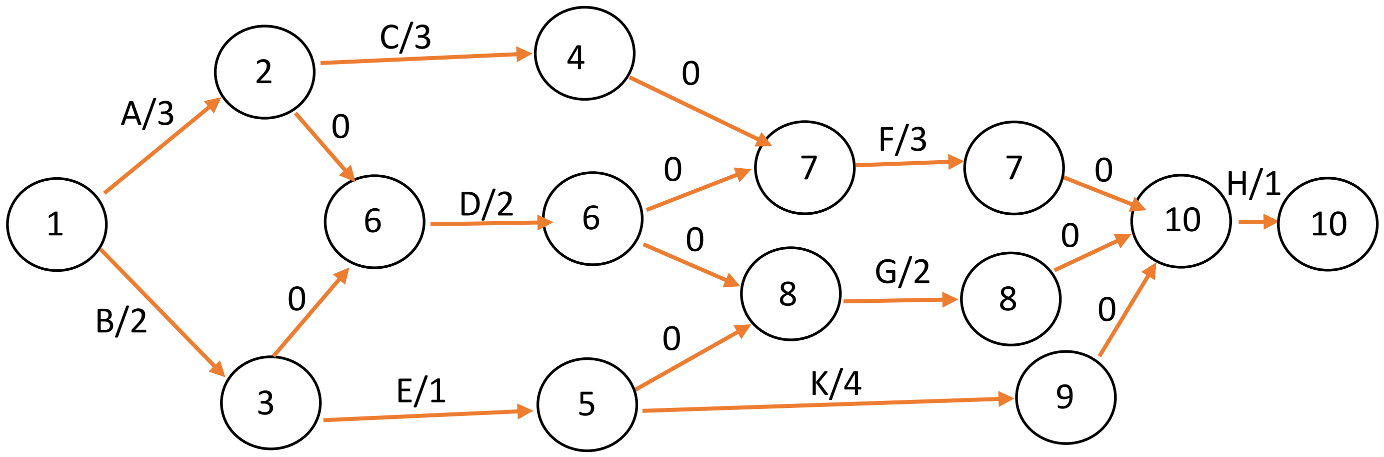
- To find the earliest completion time of the project, we merely need to find the length of the longest path from the first event to the last event.
- We can also compute the **latest time**, that each event can finish without affecting the final completion time.
- The slack time for each edge in the event node graph represents the amount of time that the completion of the corresponding activity can be delayed without delaying the overall completion.
- To perform these calculations, we convert the activity node graph to an **event node graph**.



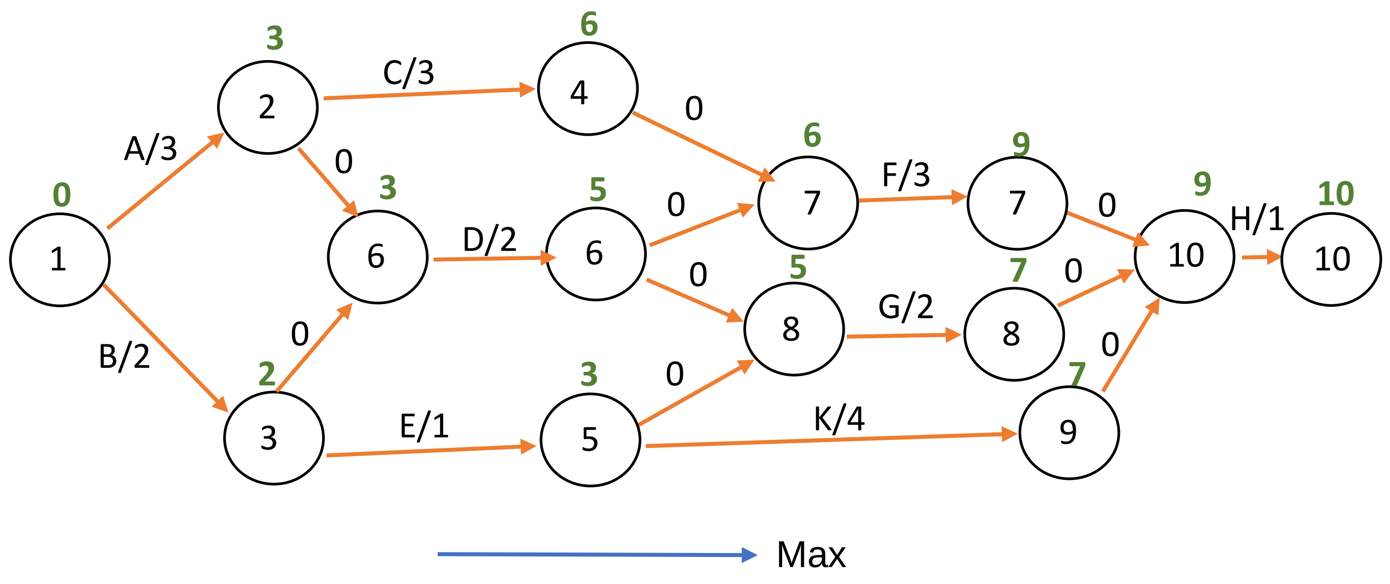
Activity node graph



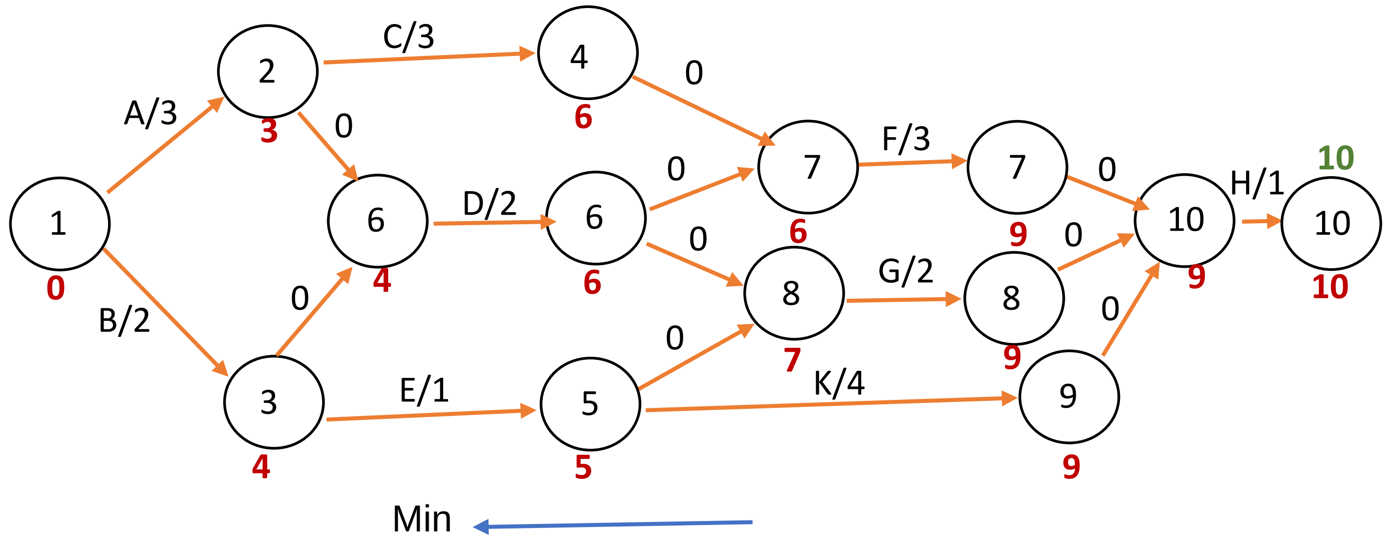
Event node graph.



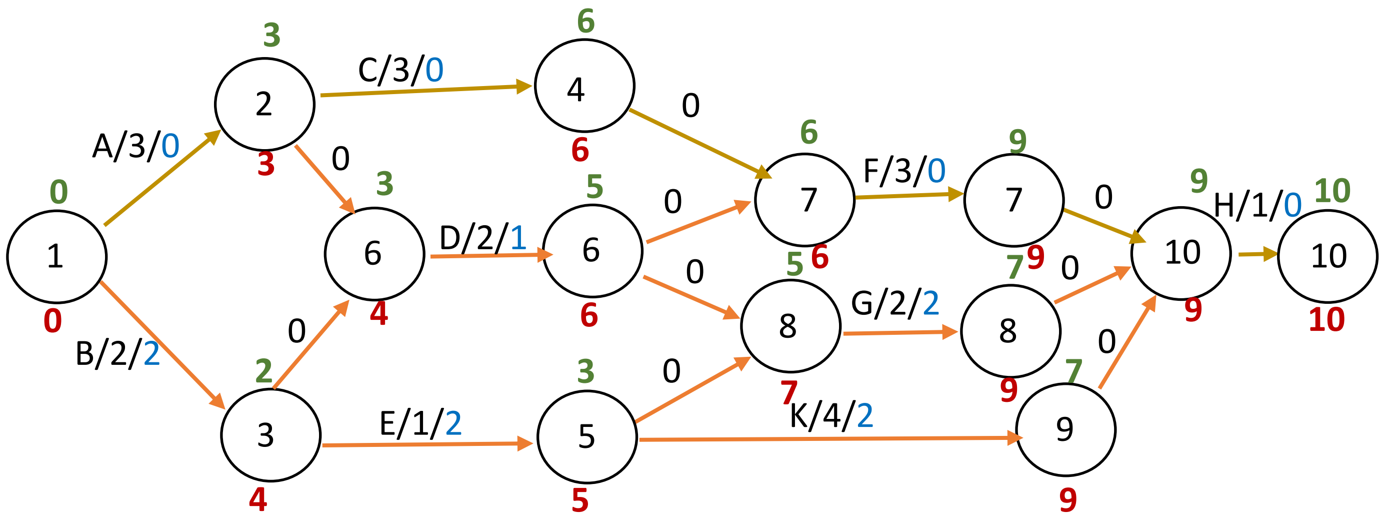
- Earliest completion times



- Latest completion times



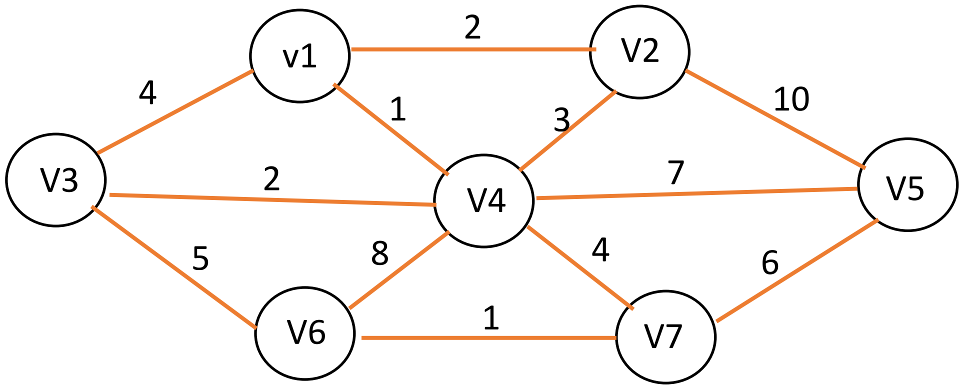
- Earliest completion times, Latest completion times and slack



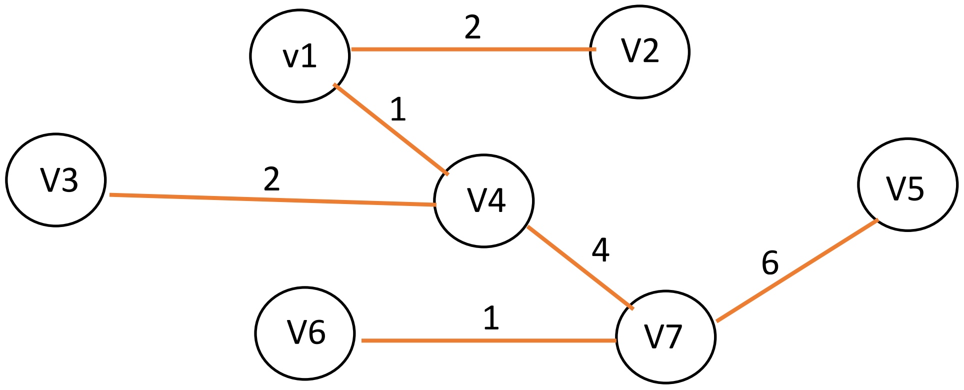
- Some activities have zero slack. These are **critical activities**, which must finish on schedule. There is at least one path considering entirely of zero slack edges, such a path is a critical path.

Minimum Spanning Tree

- A minimum spanning tree of an undirected graph G is a tree formed from graph edges that connects all the vertices of G at lowest total cost.
- A minimum spanning tree exist if and only if G is connected.



Result:



- The minimum spanning tree is a tree because it is acyclic. It is spanning because it covers every edge, and it is minimum for the obvious reason.

Example:

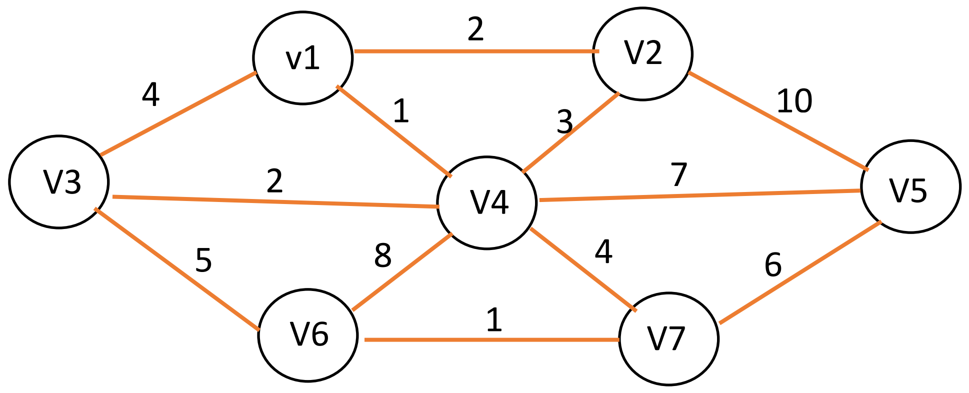
If we need to wire a house with minimum of cable, then a minimum spanning tree problem needs to be solved.

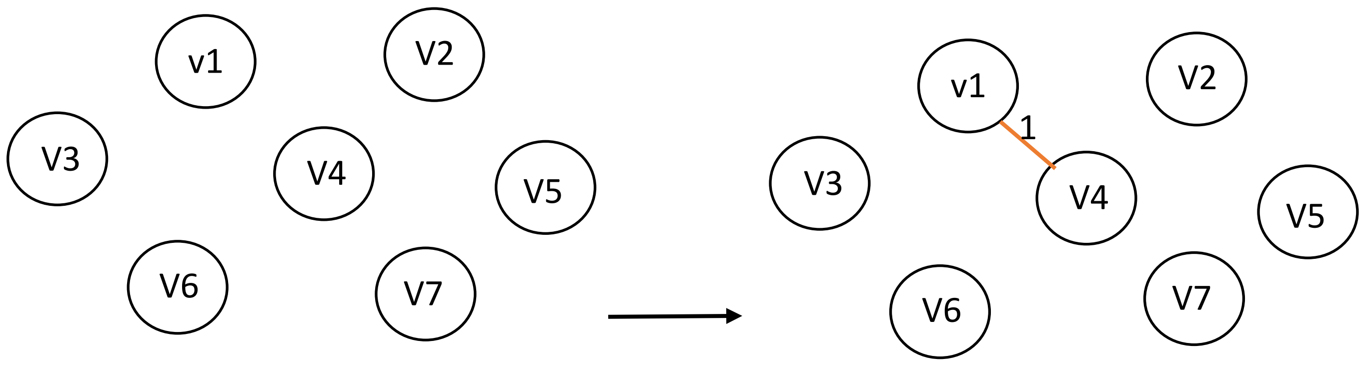
- There are two basic algorithms to solve this problem, both are greedy.

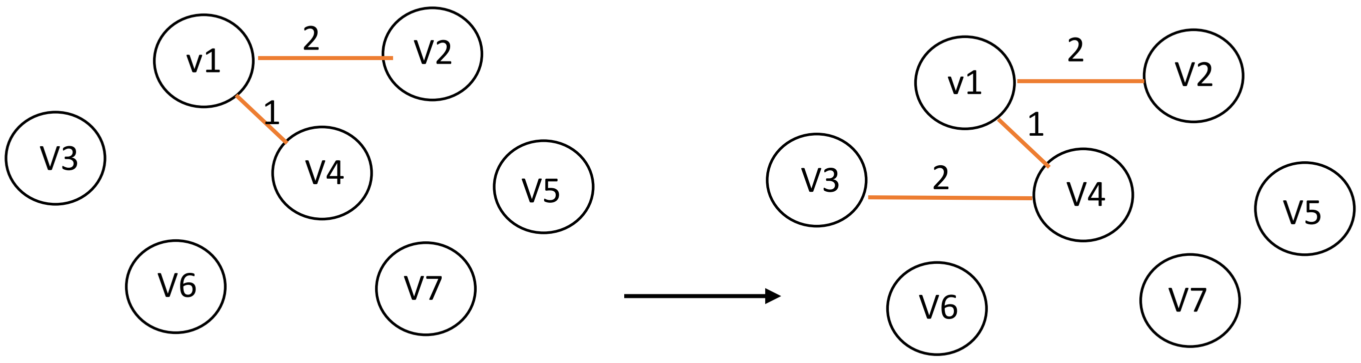
1) Prim's Algorithm

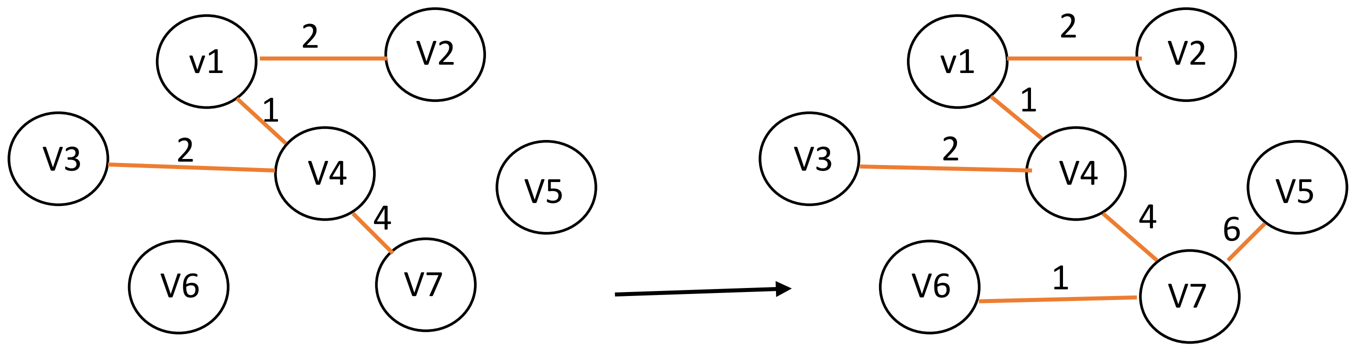
- One way to compute a minimum spanning tree it so grow the tree in successive stages. In each stage, one node is picked as the root, and we add an edge, and thus an associated vertex to the tree.
- We can see that prim's algorithm is essentially identical to Dijkstra's algorithm for shortest paths. As before, for each vertex we keep values $\mathbf{d_v}$ and $\mathbf{p_v}$ and an indication of whether it is known or unknown. $\mathbf{d_v}$ is the weight of the shortest arc connecting **v** to a known vertex, and $\mathbf{p_v}$, as before, is the last vertex to cause a change in $\mathbf{d_v}$. The rest of the algorithm is exactly the same, with the exception that since the definition of $\mathbf{d_v}$ is different, so is the update rule.

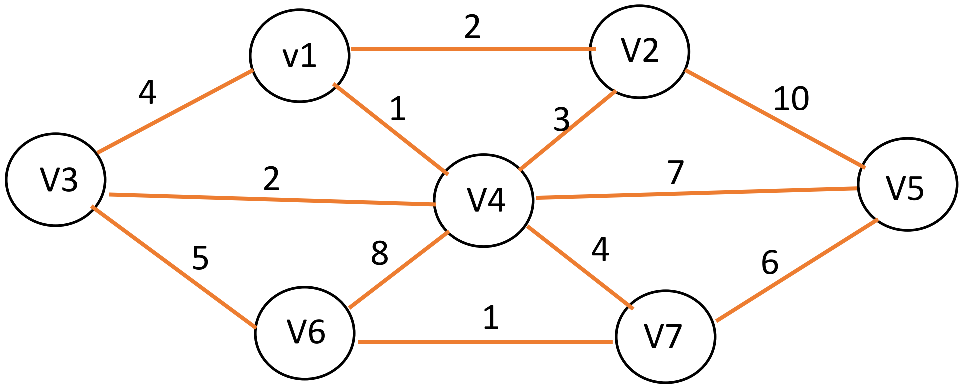
- For this problem, the update rule is even simpler than before; After a vertex v is selected, for each unknown w adjacent to v
  $d_v = \min(d_w, c_{wv})$
- The running time is $O(V^2)$ without heaps, which is optional for dense graphs, and O(E logv) using binary heaps, which is good for sparse graphs.


v1
V2
V3
V4
V5
V6
V7
2
4
1
3
10
2
7
5
8
4
6
1


v1
V2
V3
V4
V5
V6
V7
1


v1
V2
V3
V4
V5
V6
V7
2
1
2


v1
V2
V3
V4
V5
V6
V7
2
1
2
4
6
1

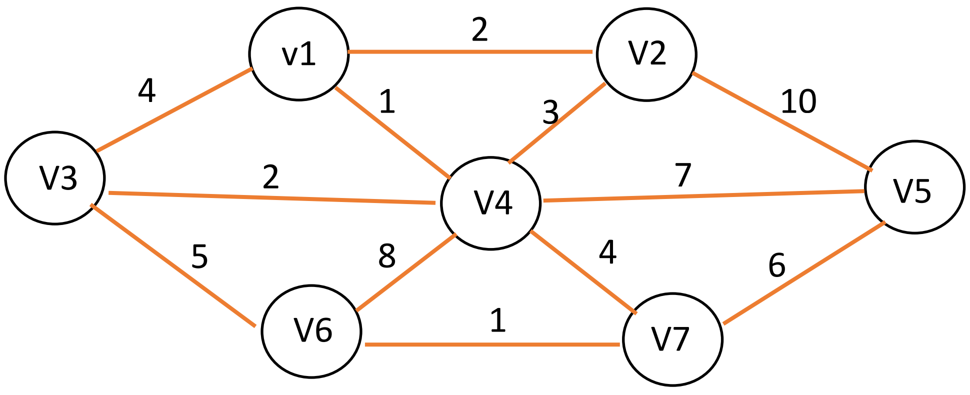


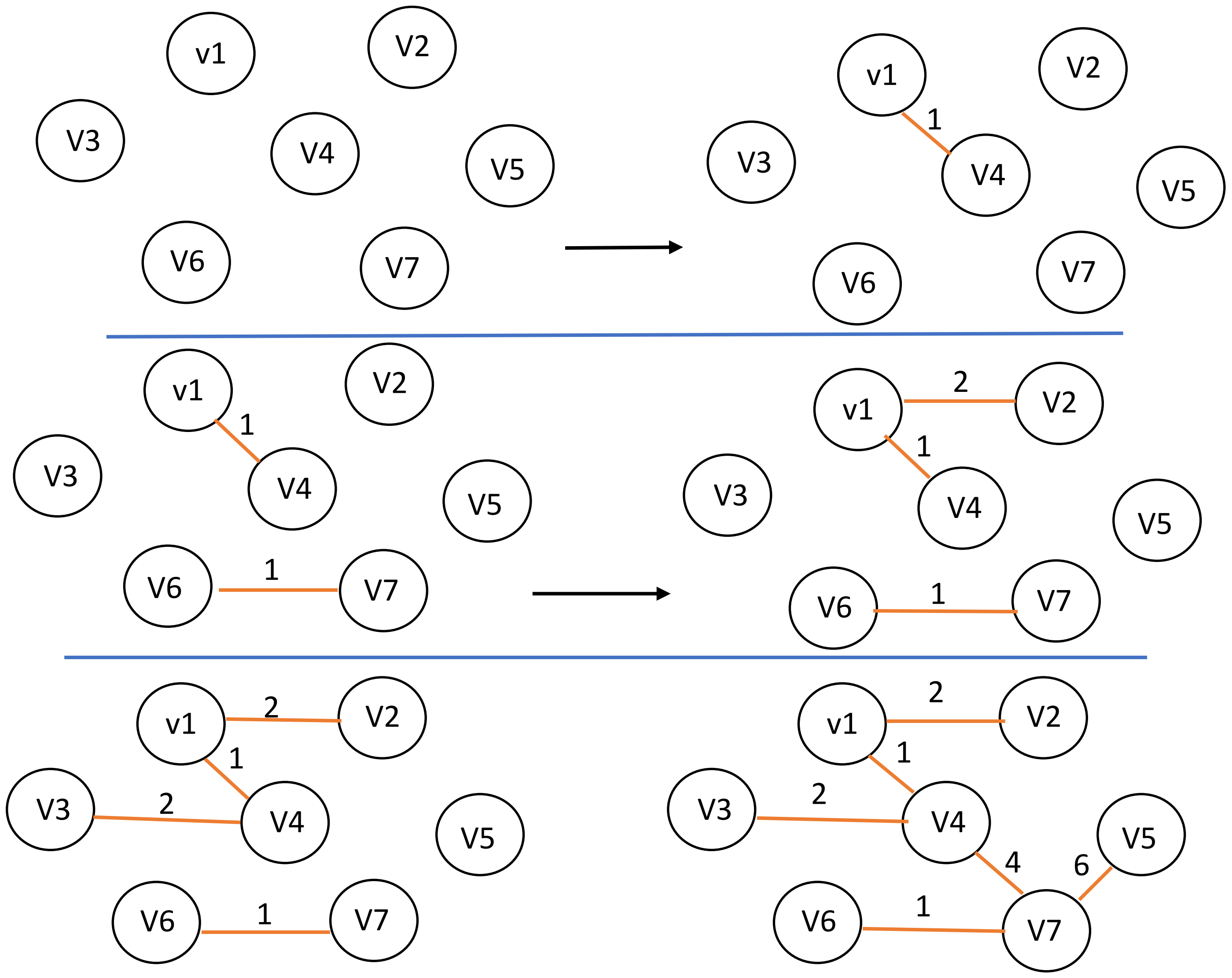
V	known	d_v	p_v	known	d_v	p_v	known	d_v	p_v	known	d_v	p_v	known	d_v	p_v	known	d_v	p_v	known	d_v	p_v
v1	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
v2	0	∞	0	0	2	v_1	0	2	v_1	1	2	v_1	1	2	v_1	1	2	v_1	1	2	v_1
v3	0	∞	0	0	4	v_1	0	2	v_4	1	2	v_4	1	2	v_4	1	2	v_4	1	2	v_4
v4	0	∞	0	0	1	v_1	1	1	v_1	1	1	v_1	1	1	v_1	1	1	v_1	1	1	v_1
v5	0	∞	0	0	∞	0	0	7	v_4	0	7	v_4	0	6	v_7	0	6	v_7	1	6	v_7
v6	0	∞	0	0	∞	0	0	8	v_4	0	5	v_3	0	1	v_7	0	1	v_7	1	1	v_7
v7	0	∞	0	0	∞	0	0	4	v_4	0	4	v_4	1	4	v_4	1	4	v_4	1	4	v_4
	Initial			**V1 is declared**			**V4 is declared**			**V2 is declared** **V3 is declared**			**V7 is declared**			**V6 is declared**			**V5 is declared**		

2) Kruskal’s Algorithm

- A second greedy strategy is continually to select the edge in order of smallest weight and accept an edge if it does not cause a cycle.
- Formally, kruskal’s algorithm maintain a forest (a collection of tree). Initially, there are **v** single node trees. Adding an edge merges two trees into one, when the algorithm terminates, there is only one tree, and this is the minimum spanning tree.
- The invariant we will use is that at any point in the process, two vertices belong to the same set of and only if they one connected in the current spanning forest.
- Thus, each vertex is initially in its own set. If **u** and **v** are in the same set, the edge is rejected, because since they are already connected, adding **(u,v)** would form a cycle. Otherwise, the edge is accepted, and a union is performed on the two sets containing **u** and **v**.
- The edges could be sorted to facilitate the selection, but building a heap in linear time is a much better idea. Then delete minimum give the edges to be tested in order.

Edge	Weight	Action
(v1, v4)	1	Accepted
(v6, v7)	1	Accepted
(v1, v2)	2	Accepted
(v3, v4)	2	Accepted
(v2, v4)	3	Rejected
(v1, v3)	4	Rejected
(v4, v7)	4	Accepted
(v3, v6)	5	Rejected
(v5, v7)	6	Accepted
…	..	…



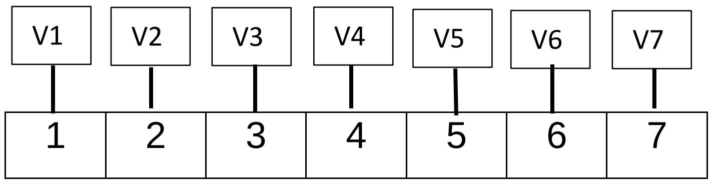


Algorithm

```
void Kruskal ( Graph G )
    int     edgesAccepted = 0;
    disjoinedSet s;
    Heap    h;
    vertex u, v;
    setType uSet, vSet;
    edge e;
    initialize ( s);
    read_Graph_into_Heap_Array( G, h);
    while ( edgesAccepted < numOfVertex )
        e = deleteMin( h );
        uSet = find( u, s);
        vSet = find( v ,s);
        if ( uSet != vSet )
            edgesAccepted ++;
            setUnion( s, uSet, vSet );
        end if
    end while
end.
```

Ideas

disjointSet s

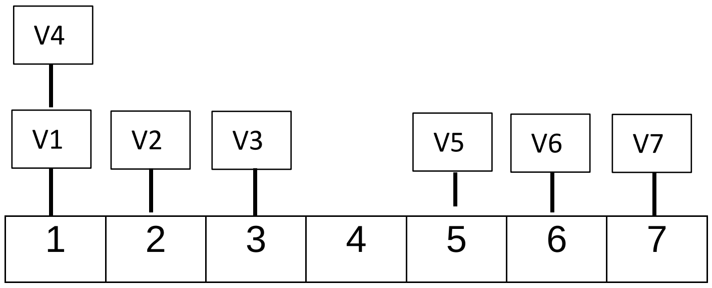


edge ➔ (v1, v4)

find (u, s); // 1

find (v, s); // 4

union (s, u, v)


V4
V1
V2
V3
V5
V6
V7
1
2
3
4
5
6
7