# Graphs

## Prim’s algorithm

Prim’s algorithm used for optimization problems, this is **Greedy** approach implementation used to find minimum spanning tree in a graph in a weighted graph.

If a graph has **V** vertices and **E** edges. The minimum spanning tree will have **V-1** edges where total cost will be minimum

* Select a vertices to start with
* Find all the nodes weight connected with the vertices & update the parent
* Find the minimum cost node from the collected in step-2 & mark it visited
* Repeat step-2 and 3 for V-1 iterations

### Implementation

1. Create 3 arrays with size V. parent, weight, visited
2. Set all the weights to Integer.MAX\_VALUE
3. Set weight [0] = 0 and parent [0] = -1 and u=0 which will represent the minimum weight node from all the listed weights**.**
4. **Loop through till V-1 times**
   1. Set visited[u] = true
   2. **Loop for all the vertices**
      1. Find the weight of connected nodes which are

* Connected to u
* Not visited
* Weight from current node < stored weight in **weight** array (may be set from other nodes)
  + 1. For match update parent and weight
  1. Update “u” i.e. set minimum weight vertices

### Other notes

Prim’s algorithm can’t work for a graph where some of the nodes are not connected.

Cycle formation won’t happen through prim’s algorithm method, which is good.

Time complexity in worst case **O (n2)**

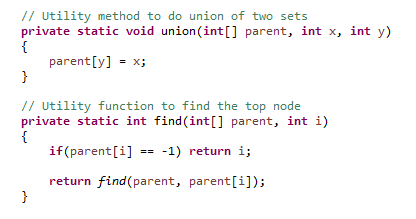
## Disjoin set

Disjoin set is a **union** **find** algorithm which is used to find cycles in Graph and heavily in **Kruskal’s** algorithm.

If an edge connecting vertices found in 2 different sets, it will union the sets to a single set. Which can be represented in form of a tree represented in array by setting parent of the vertices.

The vertices which **rank** will be considered as parent for the other vertices while making **union** in **Path compression** technique.

### Implementation



1. Create a **parent** array and fill with -1 (which says every node is their own parent)
2. Loop through all **Edges**
   1. Find parent of **src**  and **dest**
   2. If both are same then there is a **cycle**
   3. If no then make union, update the parent of one node as other, because they are connected

### Other notes

Time complexity worst case **O (n)**

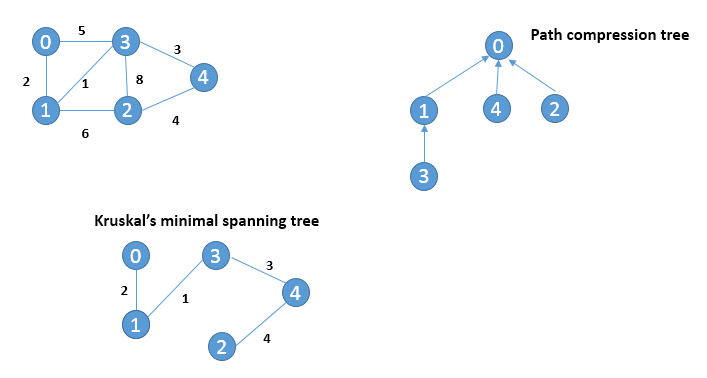
## Kruskal’s algorithm

Kruskal’s algorithm used to find minimum spanning tree in a weighted un-directed graph. Here **Greedy** approach is followed stating with minimum weight edge.

**Disjoin** set is used to avoid cycle while forming spanning tree using Kruskal’s algorithm

Again if graph have **V** vertices then number of edges will be **V-1** in spanning tree.

### Graphical representation



### Other notes

**Path compression** technique used on Disjoin set

Time complexity worst case **O (n2).** Time complexity can be improved using **Min Heap** where for every operation time complexity is **O (log n).**

## Dijkstra algorithm

Dijkstra algorithm is used to find **single source** shortest path to all the other nodes in the graph. It works on **Greedy** approach for finding minimum distance node from current node.

This approach is called **relaxation**

If (d[u] + cost(u,v) < d[v]) then d[v] = d[u] + c(v,v)

Dijkstra algorithm works for weighted directed and un-directed graph.

### Implementation

The implementation of Dijkstra is very similar to Prim’s algorithm with a bit twist.

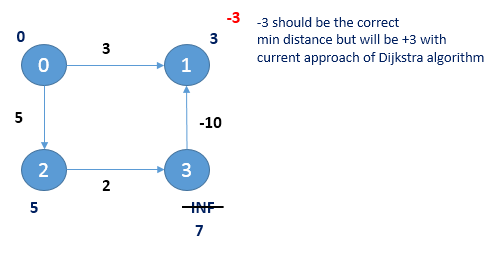
1. Create an array of distance and visited
2. Initialize all the distance value to Integer.MAX\_VALUE
3. Set distance of 0 to 0, considering 0 as the starting node.
4. Loop through all the vertices
   1. Set visited [u] to true (u- minimum weight node)
   2. Loop through all the vertices for a single “u”
      1. If there is an edge between u to v and v is not visited and distance of u + edge weight from u-v is less that current distance/weight of v
      2. Then distance[v] = distance[u] + weight(u,v)
   3. After looped through all the vertices, find the minimum weight node (i.e. u)
5. After loop over, print the solution

### Other notes

Time complexity for Dijkstra algorithm is **O (n2)** worst case.

### Where it won’t work

Dijkstra algorithm may not work properly for the graph having a **negative edge** directed graph.



# Dynamic programming

**Dynamic programming** and **Greedy method** is used for optimization problems.

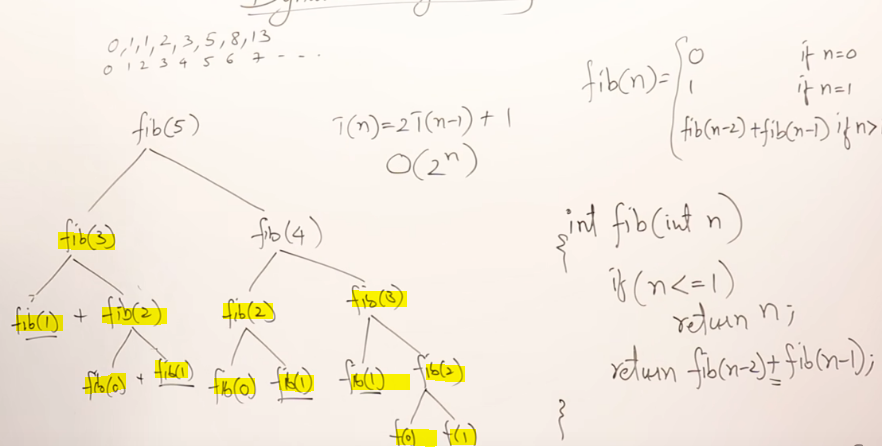
In **Greedy method** the decision is taken once and based on the decision steps would be carried out to find the optimal solution. Which also may not always guarantee the solution is optimal, but the process what got followed is optimal and based on the process that is the optimal solution.

In **Dynamic programing** the solutions will be carried out and from all the solutions the optimal solution will be picked. Here the decision will be picked on every step to approach towards an optimal solution.

The performance/time complexity of the program is improved using **Memoization** or **Tabulation**

## Memoization

Let’s take an example for finding Fibonacci series and will see what the problem was and how time complexity can be improved through Memoization.



Here if we see the highlighted once ex. **fib(3)** or **fib(2)** are computed many times and total time complexity of the program is **O (2n).**

We can improve the performance by using a global **array** where the value for pre computed fib value we will store and will use the already computed value from array rather than calling the function many times.

Using this total call will be **n+1** and hence the complexity will be **O (n+1)** i.e. **O (n)**

**Memoization** follows **top-down** approach.

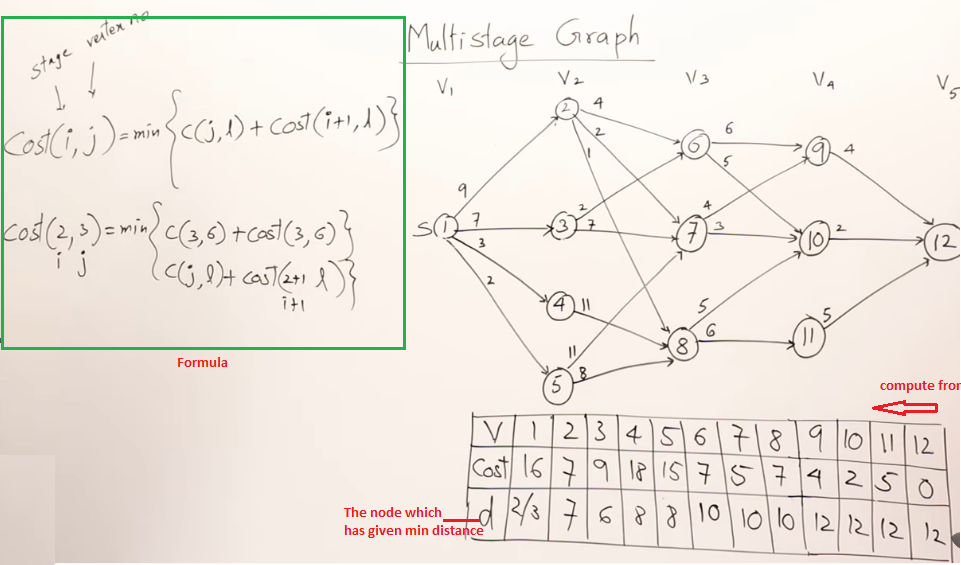
## Multistage Graph

**Multistage** graph is a directed weighted graph where the vertices are divided into stages such that the edges are connecting vertices from once stage to next stage only. First and last stage will have single vertex to represent starting and ending point of the graph.

This graph used for representing resource allocation.

We have to select a path from source to sink which gives minimum cost. As this is a minimization problem which is an optimization problem so problem can be solved using dynamic programming.

### Problem and solution

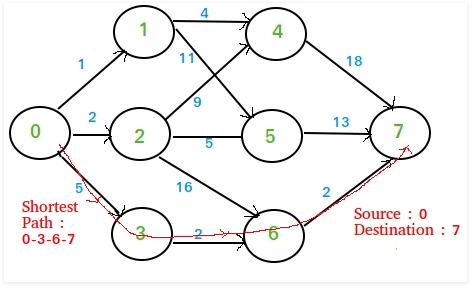


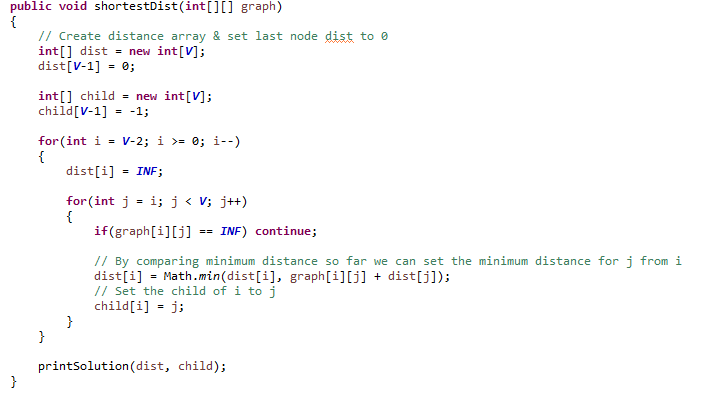
The computation happens from sink to source, but till now dynamic programming is not used. Once this part is formed we will take sequence of decisions from **top – sink** to decide the optimal path. We don’t have to compute any value and it’s already done. This method is called **Tabulation**.

While computing solution will start from **distance** array and based on it the optimal path is

**1 – 2 – 7 – 10 – 12** or **1 – 3 – 6 – 10 – 12**

1. Create a distance array & child array with size V
2. Initialize the last element of distance array to 0, as we are starting from last node
3. Initialize child array last element to -1, means no further to go.
4. Loop from last before vertex till 0th node
   1. Set the distance of node to INF
   2. Loop through j = i till V
   3. Update distance of ith node to min(dist[i], graph[i][j] + dist[j])
   4. Update child of I to j
5. Now distance [0] will give the minimum distance where from 0th node and child array will give position from 0th node to where to go.





## Floyd Warshall algorithm for all pair shortest path

**Floyd Warshall** algorithm is used to find all pair shortest path in a weighted directed graph.

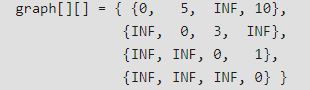
**Dijkstra** algorithm is used for single source shortest path and also can be used to find all pair shortest path by running Dijkstra on all of the nodes. But the time complexity will be **O (n2 \* n) = O (n3).**

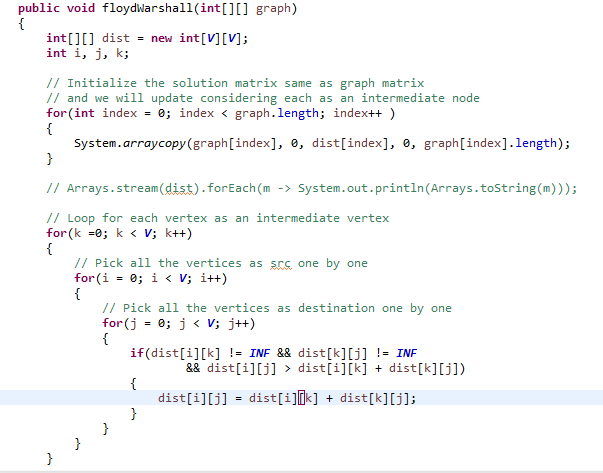
Instead we can use Floyd Warshall algorithm for finding shortest path for **all pair of vertices** using dynamic programming by considering each vertex as an intermediate node on individual step and then decision taken to find the minimum weight by considering that node as an intermediate node.

If **k** is an intermediate vertex in the shortest path from **I** to **j.** we will update the value of **dist[i][j]** when **dist[i][j] > dist[i][k] + dist[k][j]**

### Implementation

In the graph we will consider (while forming) for self-loop distance is **0** and for no edge vertices distance is **INF**





## 0/1 Knapsack problem

For given number of items **n. weight** and **profit** will be provided with a knapsack weight **M.** We need to select the items where profit will be maximized. We shouldn’t take a fraction of unit.

This is an optimization problem can be solved using through dynamic programming by taking sequence of decisions.

We need to compute all the possible arrangement and pick the best once, **Tabulation** method will be used to avoid re-computing values for overlapping sub-problems.

### Implementation

1. Loop for all the items
   1. Loop for weight ranging from 0 – M
      1. For all 0th index fill 0
      2. If weight of the item < current weight in the matrix/capacity (i.e. w)

K[i][w] = Math.max(value in the same w for last item i-1,

Current item profit + value of last item(i-1) with weight different of current K weight – weight of the item)

* + 1. If weight of the item > current bag weight capacity

Store the previous item weight in the current node cell

