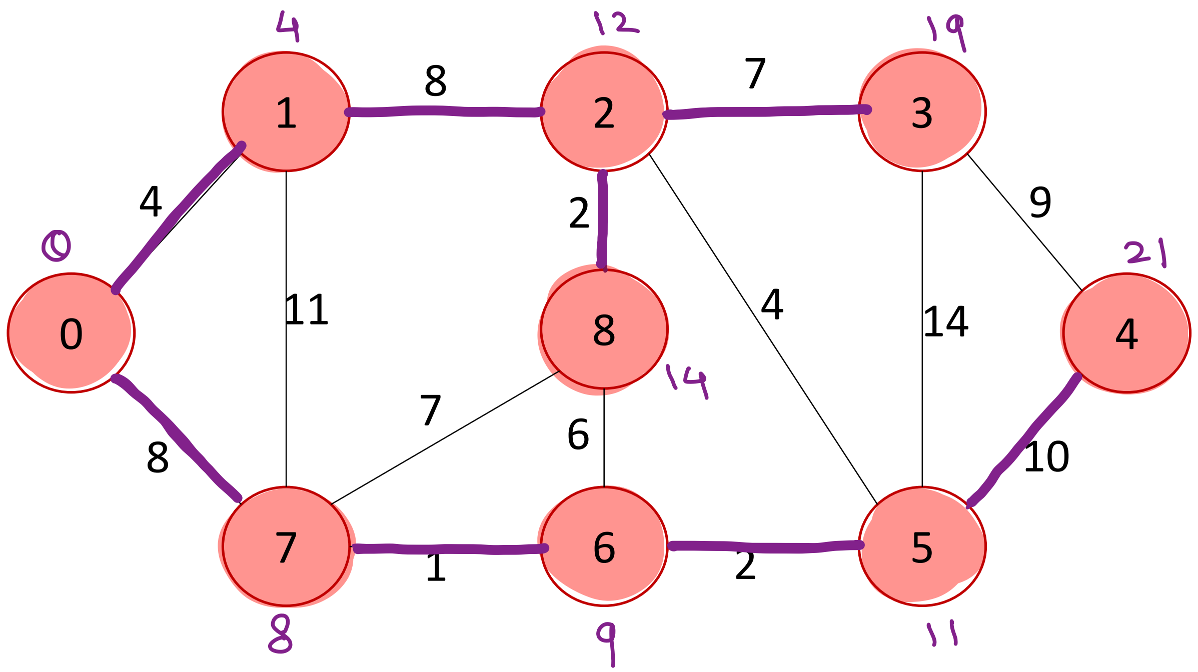
Data Structure & Algorithms

Nilesh Ghule

Dijkstra's Algorithm

- single source path length

1. Create a set *spt* to keep track of vertices included in shortest path tree.
2. Track distance of all vertices in the input graph. Distance for all vertices should be initialized to INF. The start vertex distance should be 0.
3. While *spt* doesn't include all vertices
   i. Pick a vertex u which is not there in *spt* and has minimum distance.
   ii. Include vertex u to *spt*.
   iii. Update distances of all adjacent vertices of u. For each adjacent vertex v, if distance of u + weight of edge u-v is less than the current distance of v, then update its distance as distance of u + weight of edge u-v.



Dijkstra's SPT – Analysis

1. Create a set *spt* to keep track of vertices included in shortest path tree.
2. Track distance of all vertices in the input graph. Distance for all vertices should be initialized to INF. The start vertex distance should be 0.
3. While *spt* doesn't include all vertices
   i. Pick a vertex u which is not there in *spt* and has minimum distance.
   ii. Include vertex u to *spt*.
   iii. Update distances of all adjacent vertices of u. For each adjacent vertex v, if distance of u + weight of edge u-v is less than the current distance of v, then update its distance as distance of u + weight of edge u-v.

- Time complexity (adjacency matrix)
  - V vertices: O(V)
  - get min key vertex: O(V)
  - update adjacent: O(V)
- Time complexity (adjacency matrix)
  - $O(V^2)$
- Time complexity (adjacency list)
  - V vertices: O(V)
  - get min key vertex: O(log V)
  - update adjacent: O(E) – E edges
- Time complexity (adjacency list)
  - O(E log V)

Recursion

- Function calling itself is called as recursive function.
- For each function call stack frame is created on the stack.
- Thus it needs more space as well as more time for execution.
- However recursive functions are easy to program.
- Typical divide and conquer problems are solved using recursion.
- For recursive functions two things are must
  - Recursive call (Explain process it terms of itself)
  - Terminating or base condition (Where to stop)

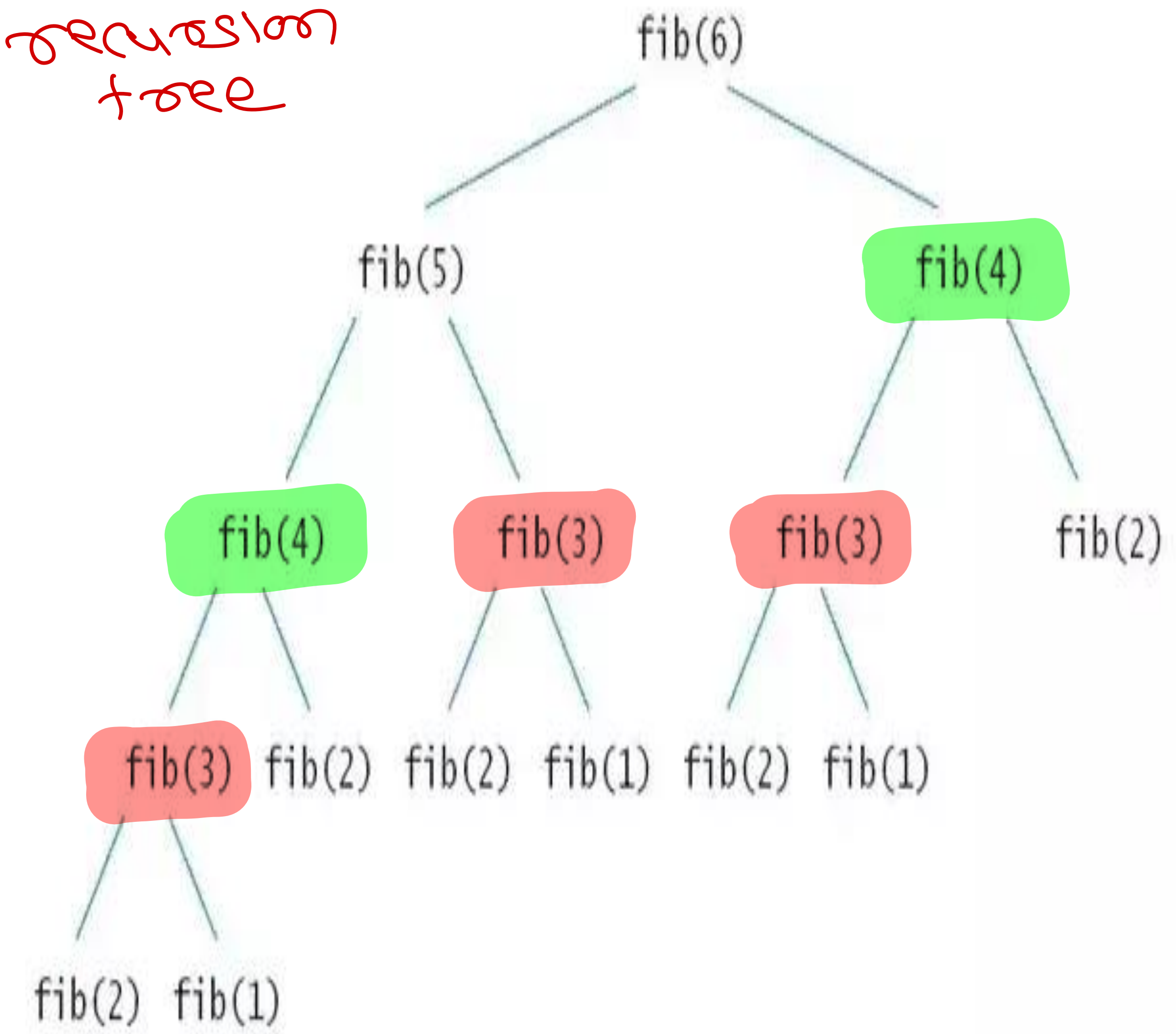
binary search
bst : preorder, postorder & inorder
bst : height
quick sort & merge sort

Recursion – Fibonacci Series

- Recursive formula
  - $T_n = T_{n-1} + T_{n-2}$
- Terminating condition
  - $T_1 = T_2 = 1$
- Overlapping sub-problem

recursion tree

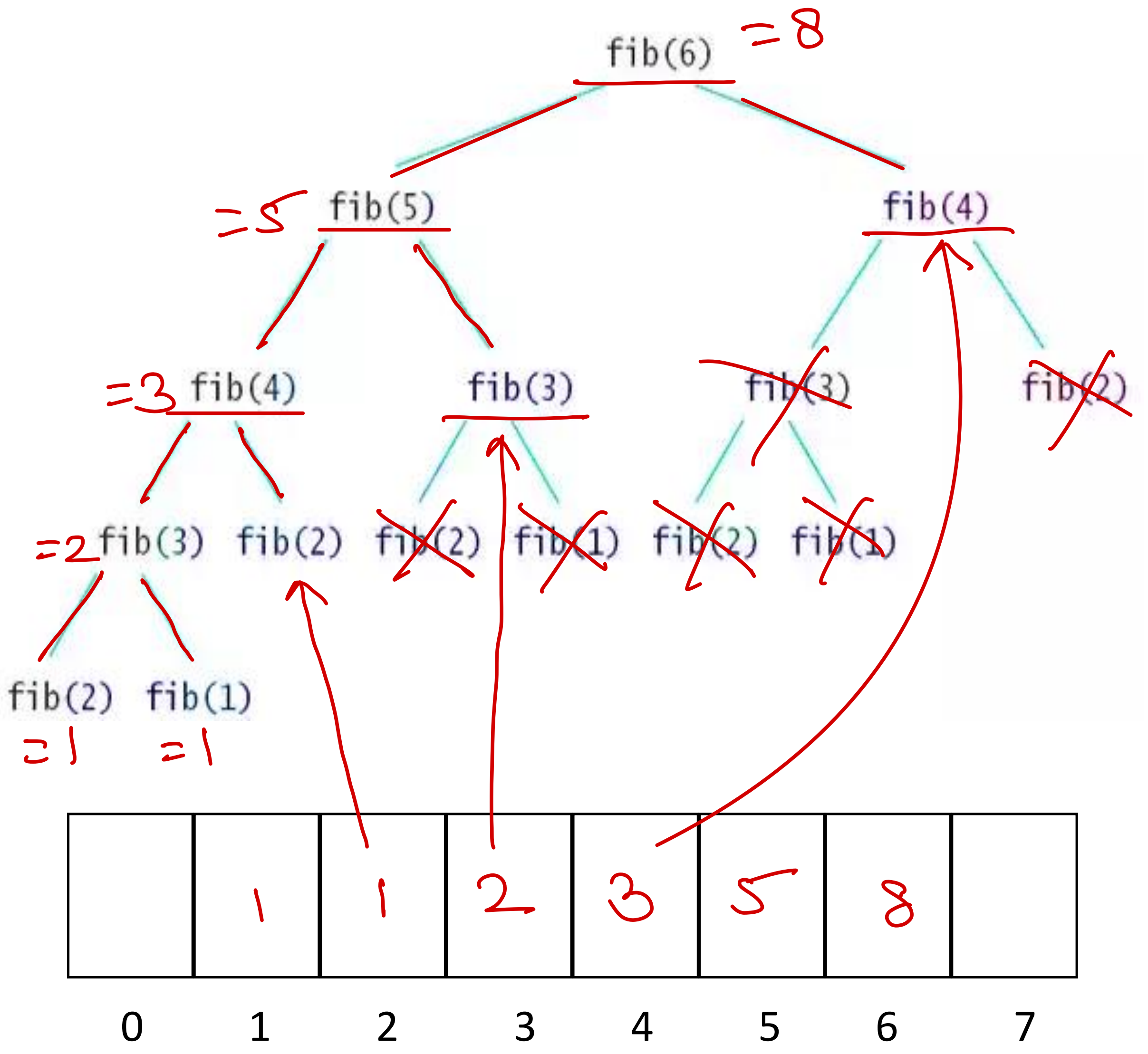
top-down approach



Memoization – Fibonacci Series

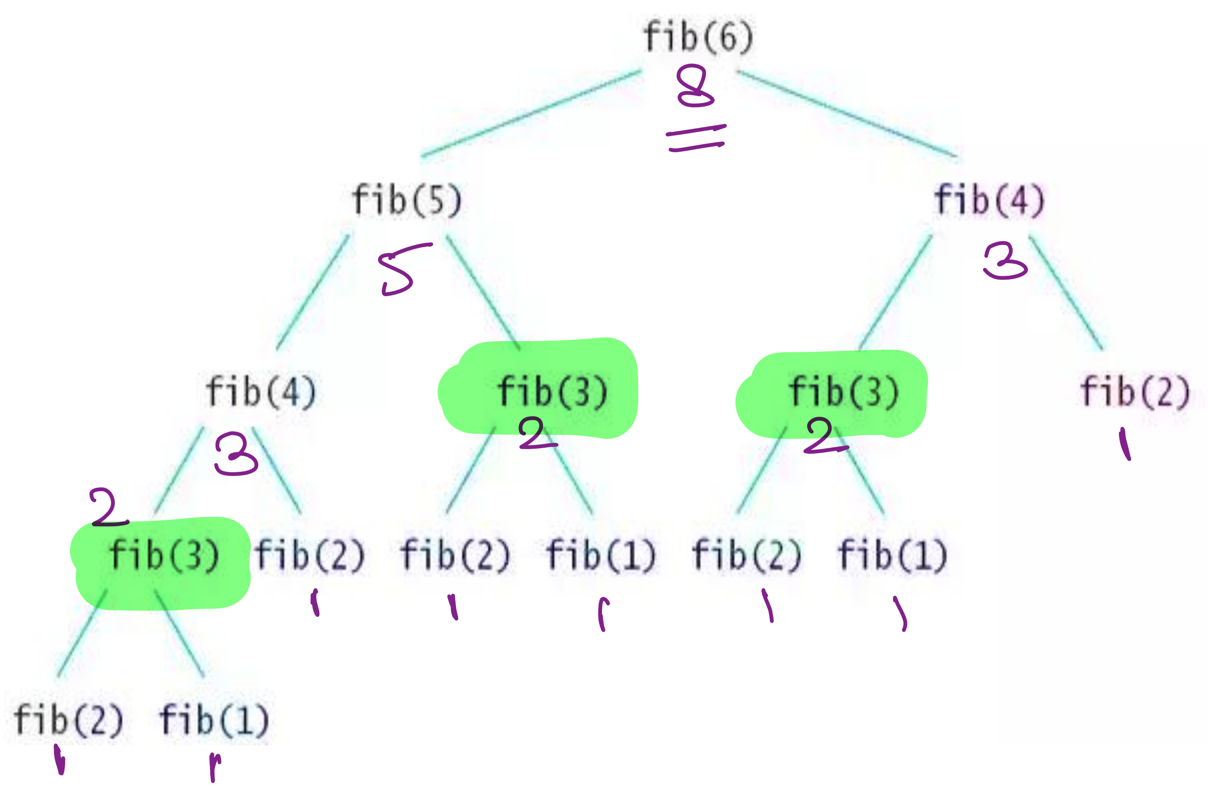
top-down approach

- It's based on the Latin word memorandum, meaning "to be remembered".
- Memoization is a technique used in computing to speed up programs.
- This is accomplished by memorizing the calculation results of processed input such as the results of function calls.
- If the same input or a function call with the same parameters is used, the previously stored results can be used again and unnecessary calculation are avoided.
- Need to rewrite recursive algorithm. Using simple arrays or map/dictionary.



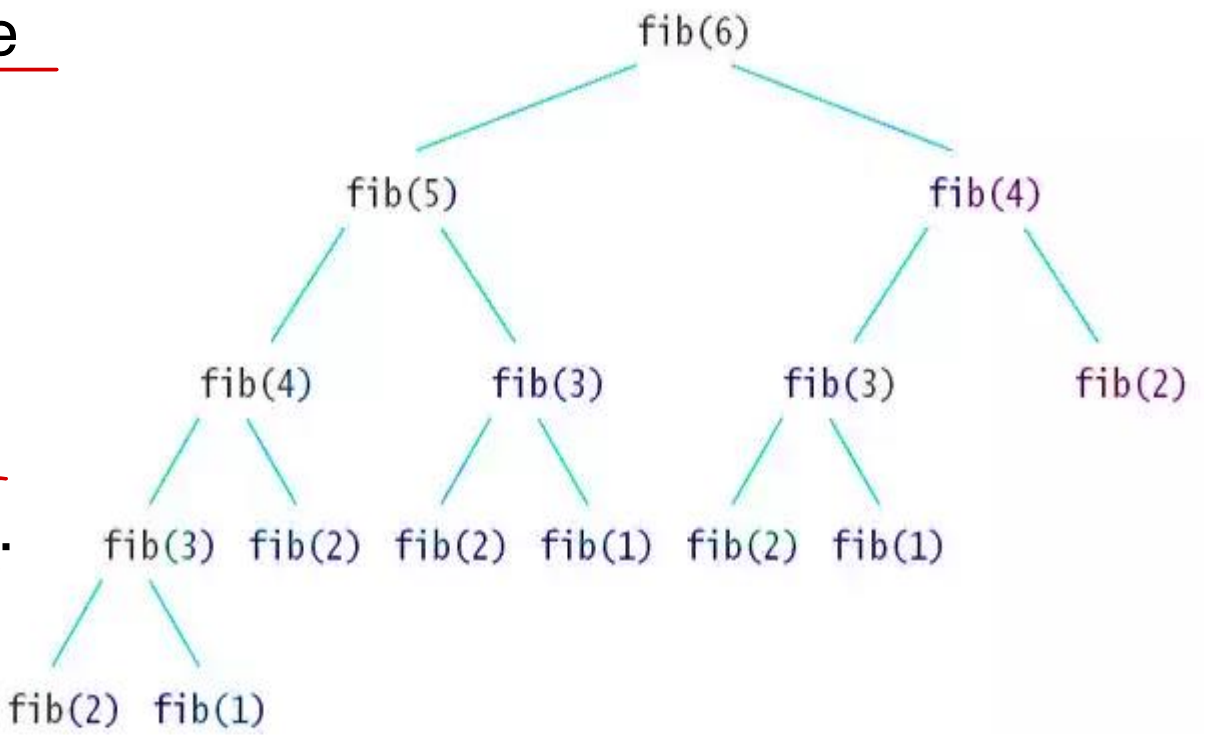
Dynamic Programming

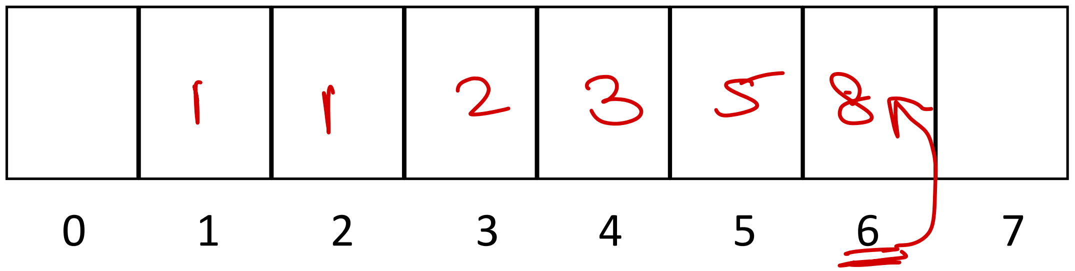
- Dynamic programming is another optimization over recursion.
- Typical DP problem give choices (to select from) and ask for optimal result (maximum or minimum).
- Technically it can be used for the problems having two properties
  - Overlapping sub-problems
  - Optimal sub-structure
- To solve problem, we need to solve its sub-problems multiple times.
- Optimal solution of problem can be obtained using optimal solutions of its sub-problems.



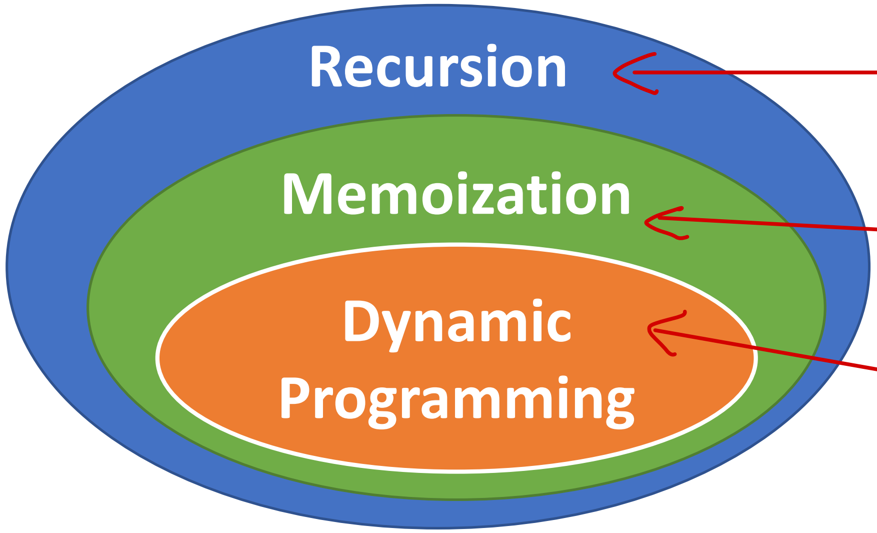
Dynamic Programming – Fibonacci Series

- Alternative solution to DP is memoizing the recursive calls. This solution needs more stack space, but similar in time complexity.
- Memoization is also referred as top-down approach.
- DP solution is bottom-up approach.
- DP use 1-d array or 2-d array → or map/dict to save state.
- Greedy algorithms pick optimal solution to local problem and never reconsider the choice done.
- DP algorithms solve the sub-problem in a iteration and improves upon it in subsequent iterations.





Dynamic Programming



- Top down
  - → divide & conquer
  - → overlapping sub-problem
- Optimized Top down
  - overlapping sub-problem
- Bottom up
  - overlapping sub-problem

r r a d a r

start ↓ / end →	0	1	2	3	4	5
0	✓	✓				
1		✓				✓
2			✓		✓	
3				✓		
4					✓	
5						✓

← end

↑ Start

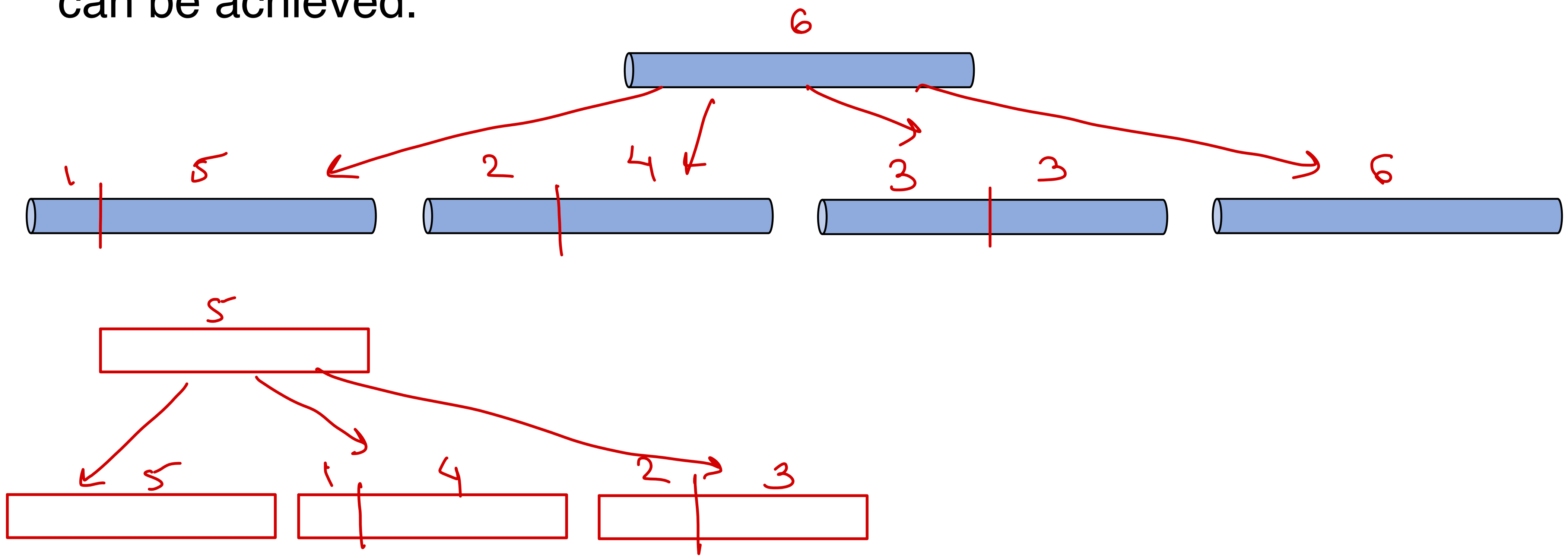
$$\overset{1}{str[start]} == \overset{5}{str[end]}$$

$$r == r$$

$$mat[\underbrace{start+1}_{2}][\underbrace{end-1}_{4}] = \underline{true}$$

Dynamic Programming

- Rod cutting problem: Cut the rod of given price so that maximum price can be achieved.

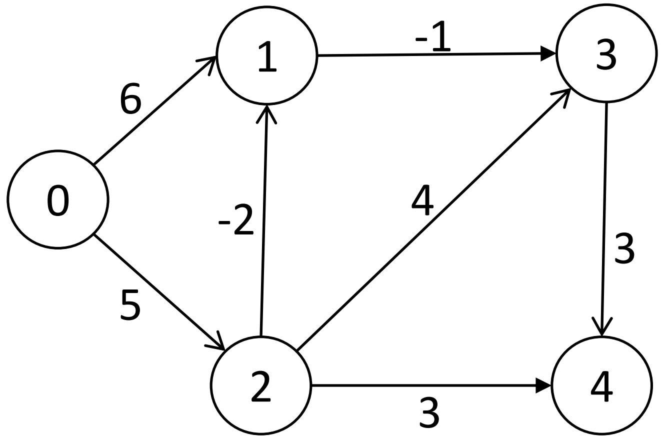


Length	Price
1	1
2	5
3	8
4	9
5	10
6	14
7	17
8	20
9	24
10	30

Bellman Ford Algorithm

single source path length - weighted graph
Dijkstra's algo cannot work with -ve edges.

- Initializes distances from the source to all vertices as infinite and distance to the source itself as 0.
- Calculates shortest distance V-1 times: For each edge u-v, if dist[v] > dist[u] + weight of edge u-v, then update dist[v], so that dist[v] = dist[u] + weight of edge u-v.
- Check if negative edge in the graph: For each edge u-v, if dist[v] > dist[u] + weight of edge uv, then graph has –ve weight cycle.



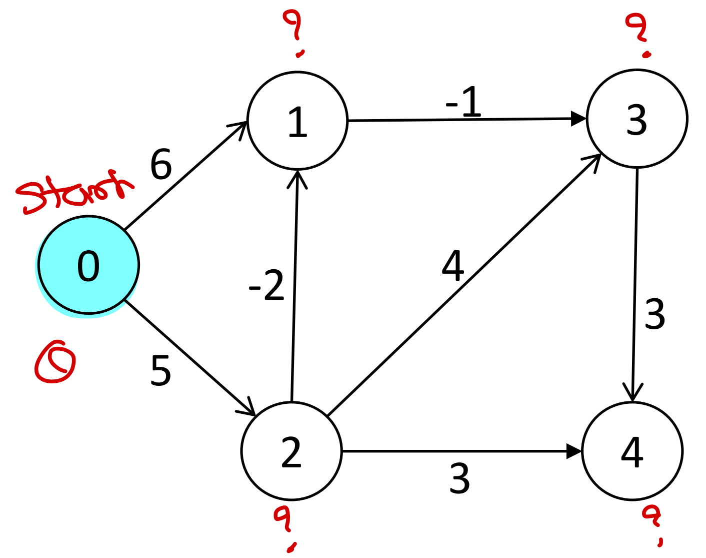
Src	Des	Wt
3	4	3
2	4	3
2	3	4
2	1	-2
1	3	-1
0	2	5
0	1	6

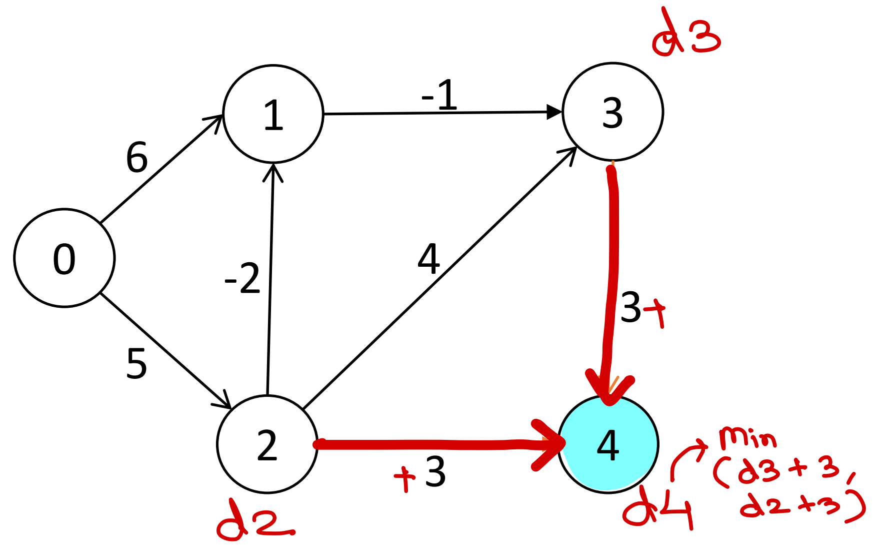
graph with -ve edges

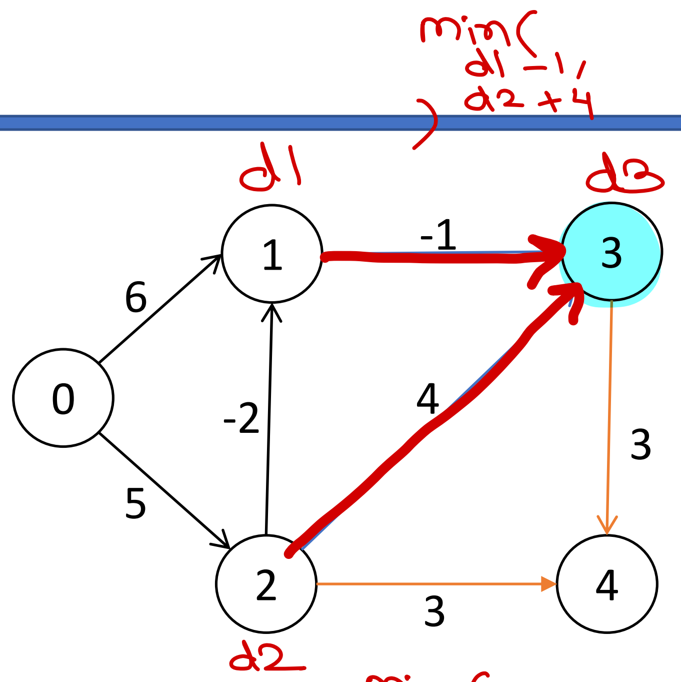
① electronic cct
↳ current with direction ↑↓

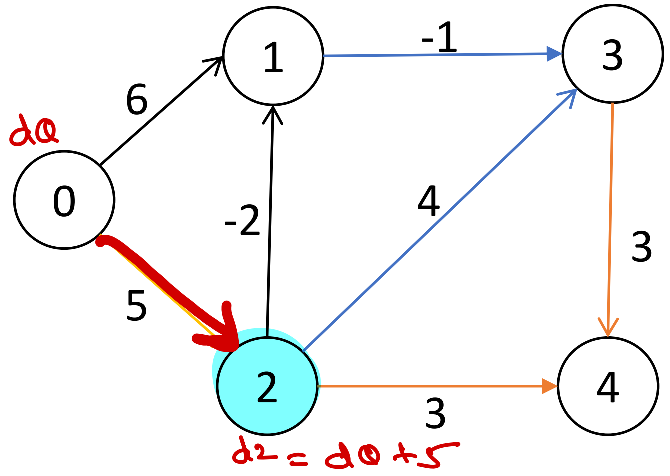
② chemical reactions
↳ energy / heat ⇄

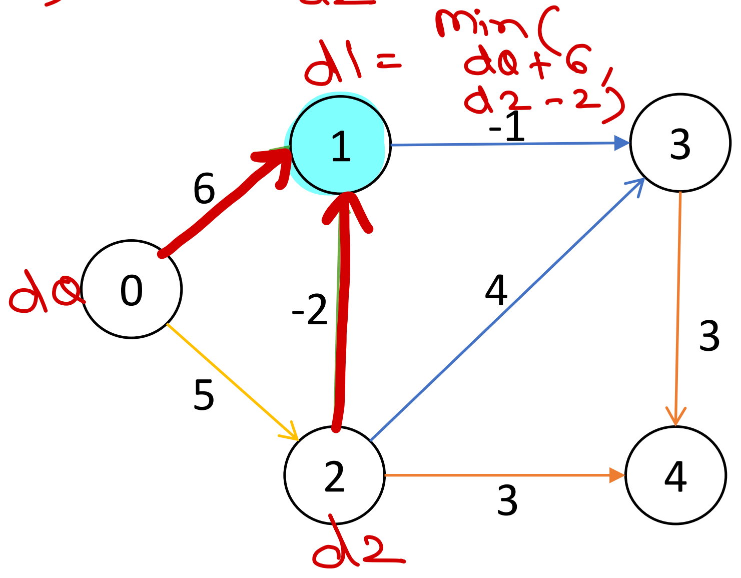
Bellman Ford Algorithm







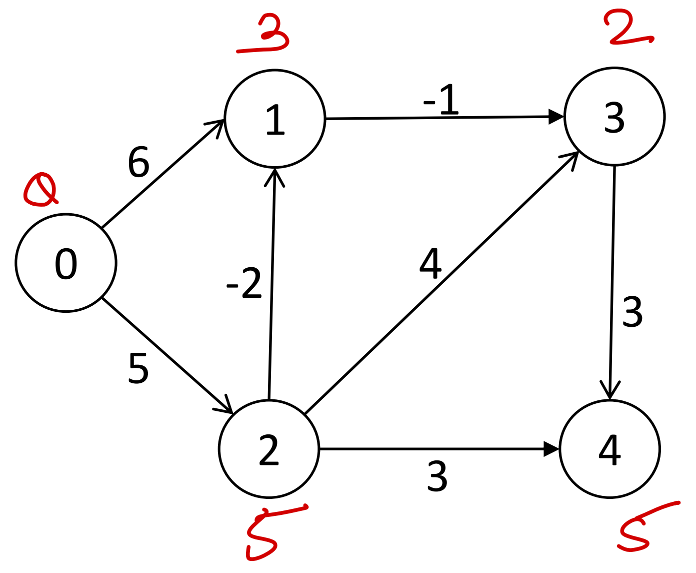




Bellman Ford Algorithm

V - 1 Passes

	0	1	2	3	4
Pass 0	0	∞	∞	∞	∞
Pass 1	0	6	5	∞	∞
Pass 2	0	3	5	2	8
Pass 3	0	3	5	2	5
Pass 4	0	3	5	2	5



	$\overset{u}{\textbf{Src}}$	$\overset{v}{\textbf{Dest}}$	**Wt**
✓	3	4	3
✓	2	4	3
✓	2	3	4
✓	2	1	-2
✓	1	3	-1
✓	0	2	5
✓	0	1	6

Bellman Ford cannot work with -ve weight cycles.
In this case dist of vertices keep changing in each iteration.

$$\text{if}(\text{dist}[u] + \text{wt}(u,v) < \text{dist}[v])$$
$$\text{dist}[v] = \text{dist}[u] + \text{wt}(u,v)$$

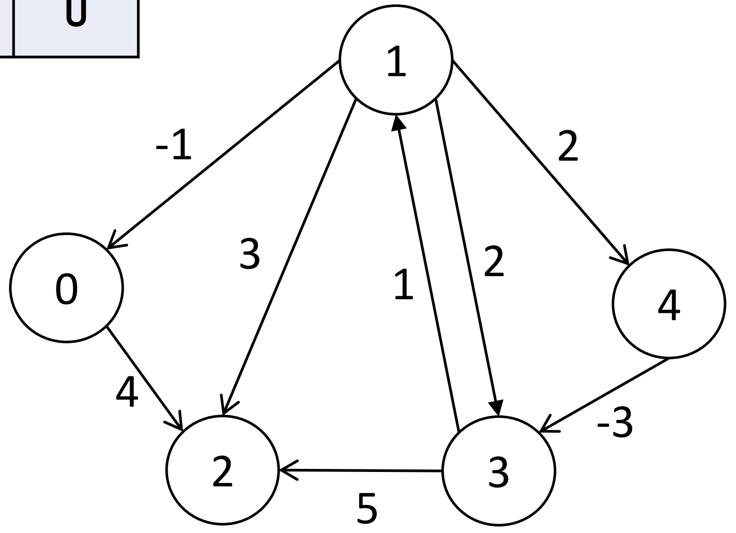
Warshall Floyd Algorithm

- Algorithm

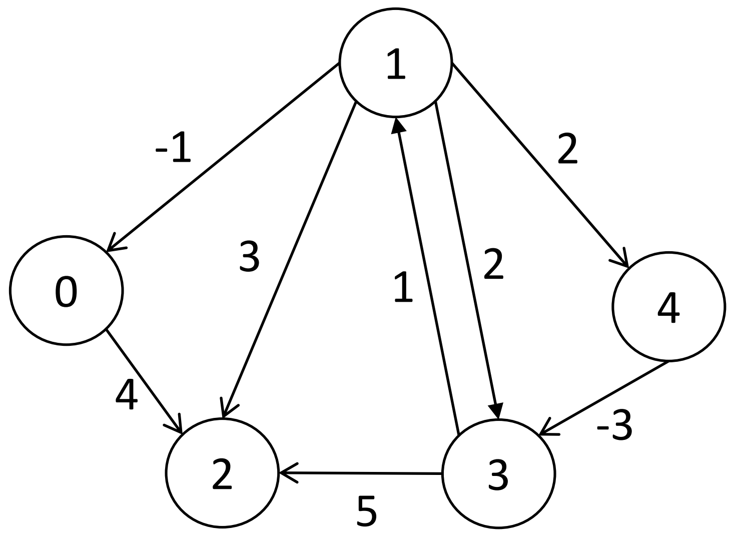
1. Create distance matrix to keep distance of every vertex from each vertex. Initially assign it with weights of all edges among vertices (i.e. adjacency matrix).

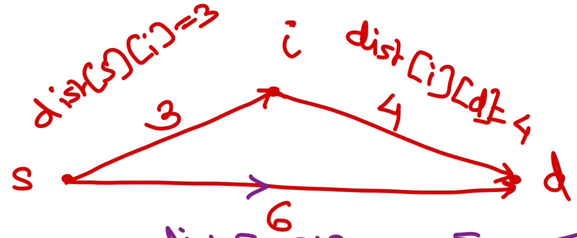
2. Consider each vertex (i) in between pair of any two vertices (s, d) and find the optimal distance between s & d considering intermediate vertex i.e. dist(s,d) = dist(s,i) + dist(i,d), if dist(s,i) + dist(i,d) < dist(s,d).

	0	1	2	3	4
0	0	∞	4	∞	∞
1	-1	0	3	2	2
2	∞	∞	0	∞	∞
3	∞	1	5	0	∞
4	∞	∞	∞	-3	0



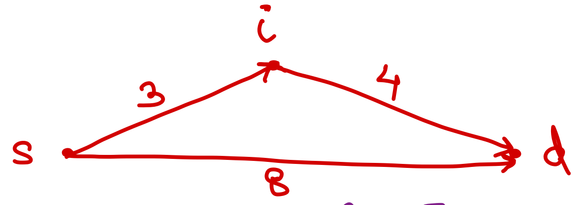
Warshall Floyd Algorithm





$dist[s][d] < dist[s][i] + dist[i][d]$

$dist[s][d] = 6$



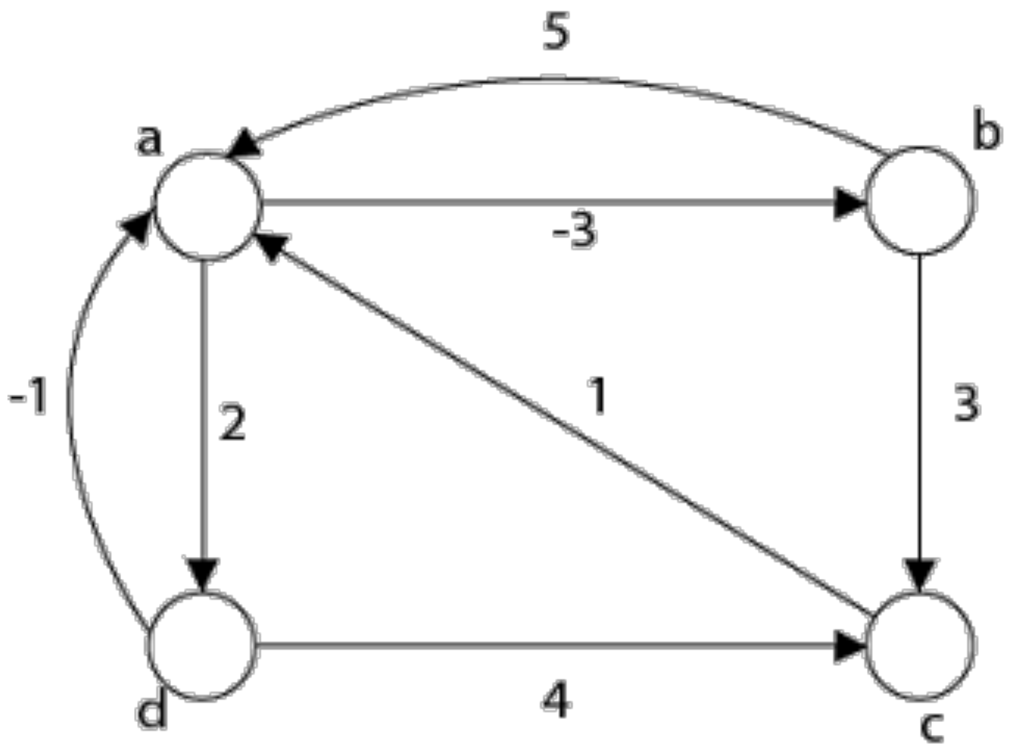
$dist[s][d] > dist[s][i] + dist[i][d]$

$dist[s][d] = 7$

Johnson's Algorithm – all pair shortest path also.

- Time complexity of Warshall Floyd is O(V^3).
- Applying Dijkstra's algorithm on V vertices will cause time complexity O(V * V log V). This is faster than Warshall Floyd.
- However Dijkstra's algorithm can't work with –ve weight edges.
- Johnson use Bellman ford to reweight all edges in graph to remove –ve edges. Then apply Dijkstra to all vertices to calculate shortest distance. Finally reweight distance to consider original edge weights.
- Time complexity of the algorithm:

  O(VE + V^2 log V).



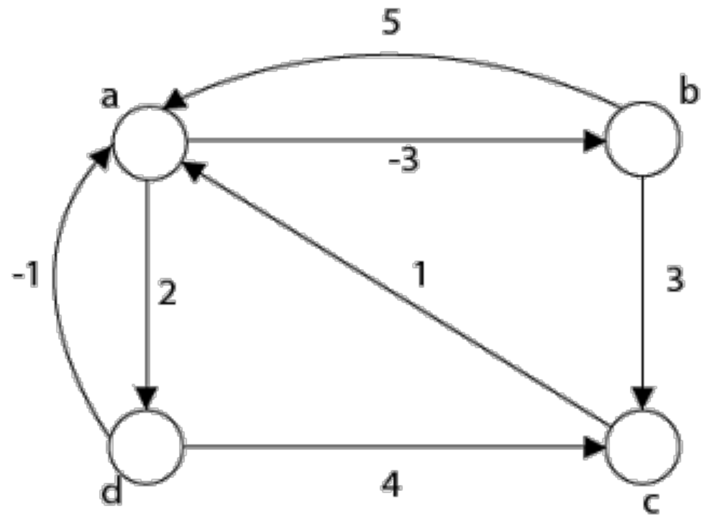
Johnson's Algorithm

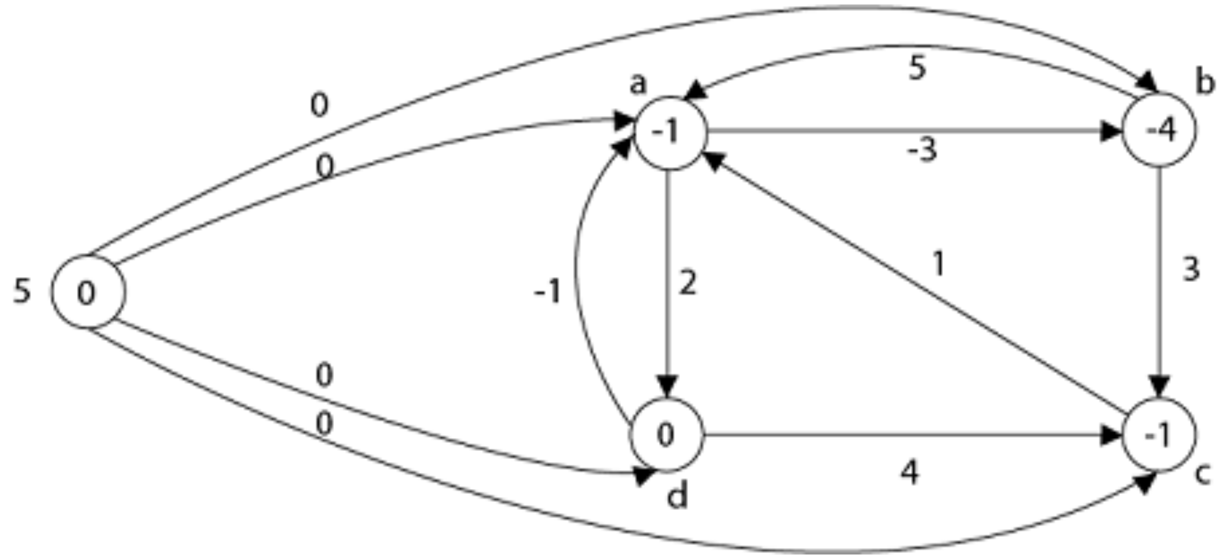
1. Add a vertex (s) into a graph and add edges from it all other vertices, with weight 0.
2. Find shortest distance of all vertices from (s) using Bellman Ford algorithm.

   a = -1, b = -4, c = -1, d = 0 and s = 0
3. Reweight all edges (u, v) in the graph, so that, they become non negative.

   weight(u, v) = weight(u, v) + d(u) – d(v)

- w(a, b) = 0
- w(b, a) = 2
- w(b, c) = 0
- w(c, a) = 1
- w(d, c) = 5
- w(d, a) = 0
- w(a, d) = 1



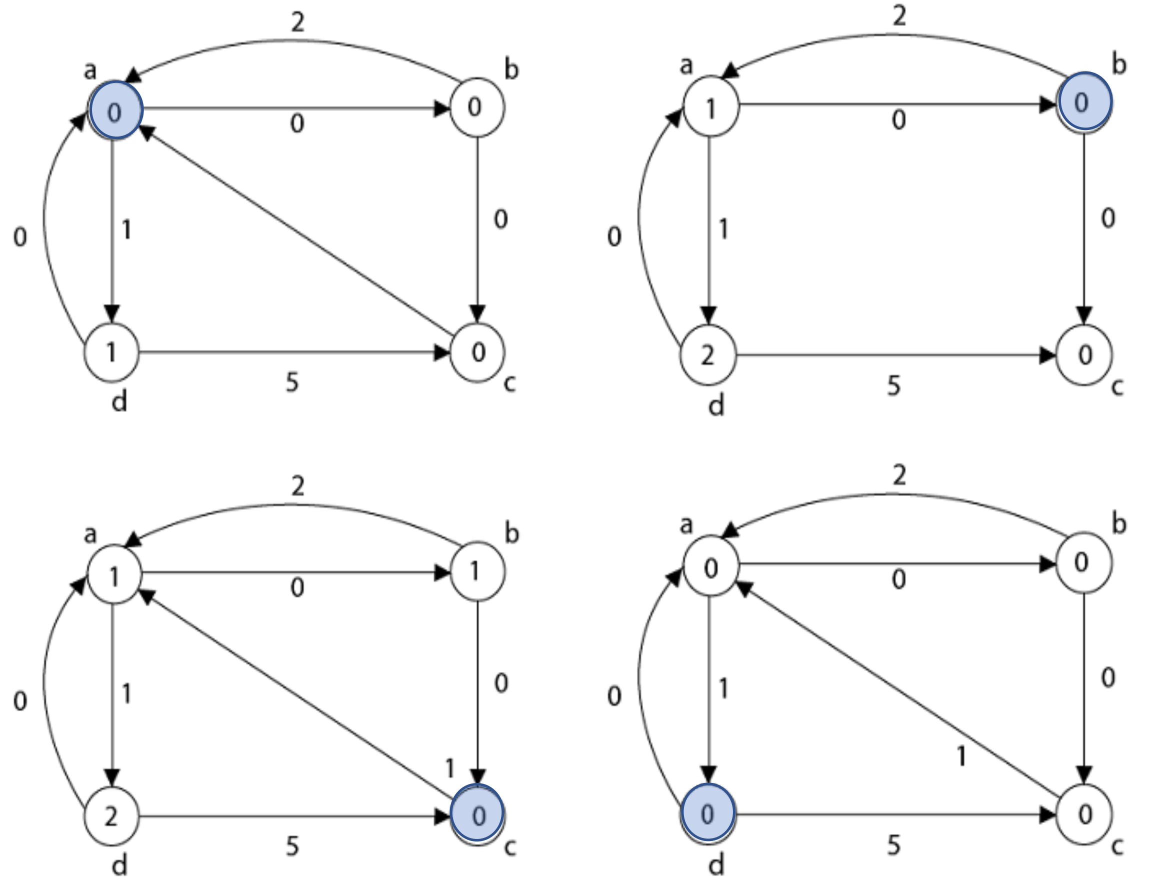


Johnson’s Algorithm

4. Apply Dijkstra on each vertex to calculate shortest distance to all other vertices.
5. Reweight all distances to consider original weights.

$$\text{dist}(u, v) = \text{dist}(u, v) + d(v) - d(u)$$

	a	b	c	d
a	0 -> 0	0 -> -3	0 -> 0	1 -> 2
b	1 -> 4	0 -> 0	0 -> 3	2 -> 6
c	1 -> 1	1 -> -2	0 -> 0	2 -> 3
d	0 -> -1	0 -> -4	0 -> -1	0 -> 0



A* Search Algorithm

- Point to point approximate shortest path finder algorithm.
- This algorithm is used in artificial intelligence.
- Commonly used in games or maps to find shortest distance in faster way.
- It is modification of BFS.
- Put selected adjacent vertices on queue, based on some heuristic.
- A math function is calculated for vertices
  - f(v) = g(v) + h(v) → vertex with min f(v) is picked.
  - g(v) → cost of source to vertex v
  - *h(v)* → estimated cost of vertex v to destination.

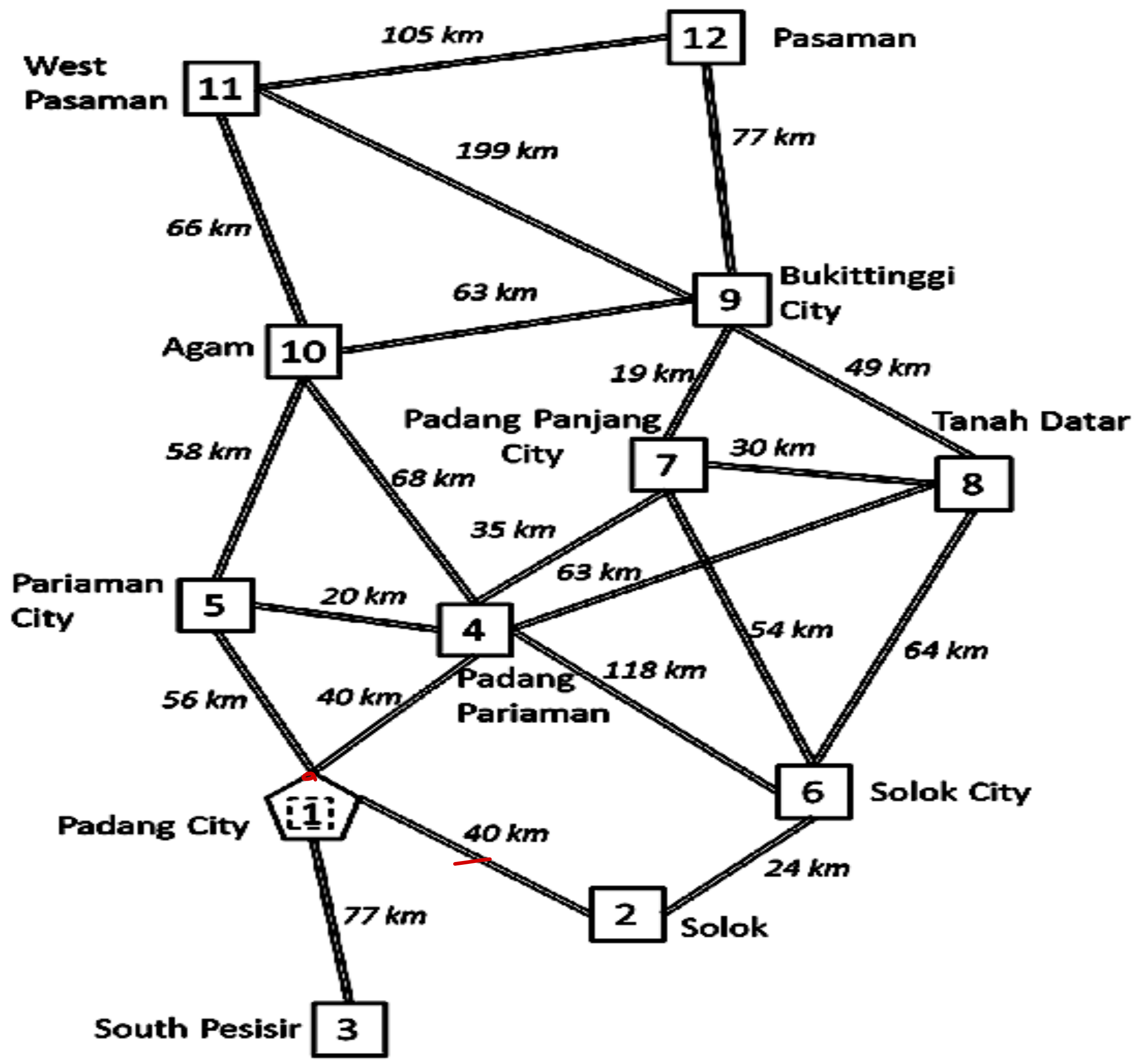
	0	1	2	3	4	5	6	7	8	9
0	1	0	1	1	1	1	0	1	1	1
1	1	1	1	0	1	1	1	0	1	1
2	1	1	1	0	1	1	0	1	0	1
3	0	0	1	0	1	0	0	0	0	1
4	1	1	1	0	1	1	1	0	1	0
5	1	0	1	1	1	1	0	1	0	0
6	1	0	0	0	0	1	0	0	0	1
7	1	0	1	1	1	1	0	1	1	1
8	1	1	1	0	0	0	1	0	0	1
9	1	0	1	0	0	0	0	0	0	1

Graph applications

- Maps uses graphs for showing routes and finding shortest paths. Intersection of two (or more) roads is considered as vertex and the road connecting two vertices is considered to be an edge.

all - pair shortest path

src	1	2	3	4	5	6	7	8	9	10	11
1	0	40	77								
→ 2	40	0	117								
3											
4											
5											
6											
7											
8											
9											
10											
11											



SUNBEAM

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>