Graph algorithms

- **Graph Traversal (Graph Searching)**
  - Breadth-first search
  - Depth-first search
- **Shortest-Path Algorithm**
  - Dijkstra's algorithm
- **Minimum Spanning Tree**
  - Prim's Algortihm
  - Kruskal's Algorithm

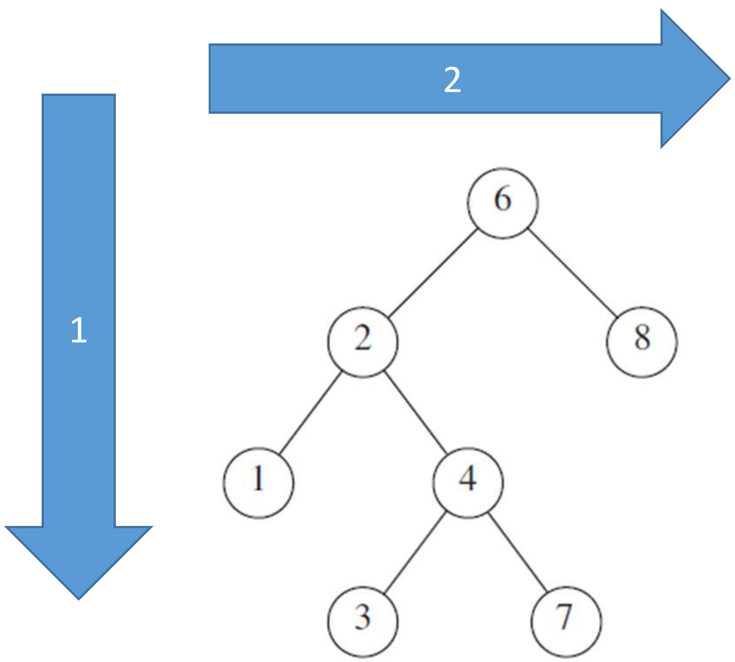
Graph Traversals

- Traversing a graph means visiting each vertex of the graph exactly once.

- Two common graph traversal algorithms
  - ***Depth first search*** (DFS)
  - ***Breadth first search*** (BFS)

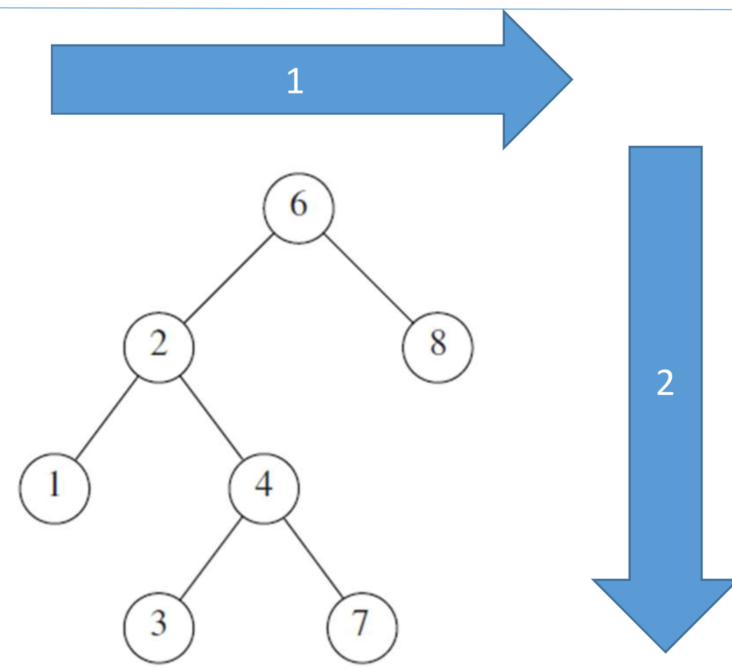
TREE

Depth First Search



6 2 1 4 3 7 8

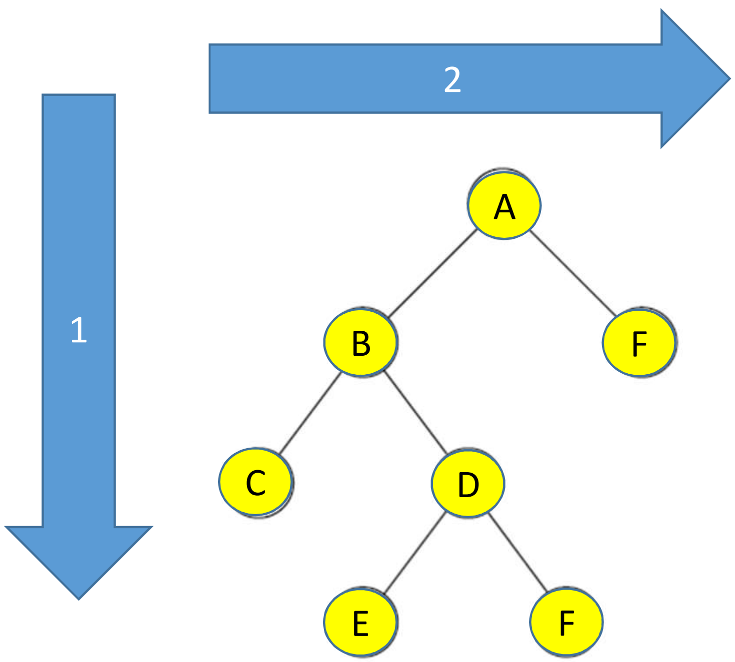
Breadth First Search



6 2 8 1 4 3 7

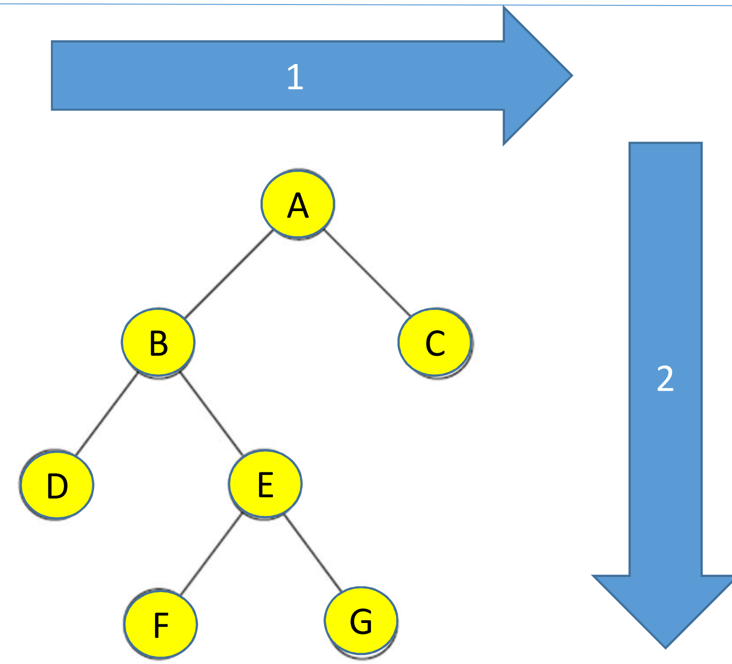
TREE

Depth First Search



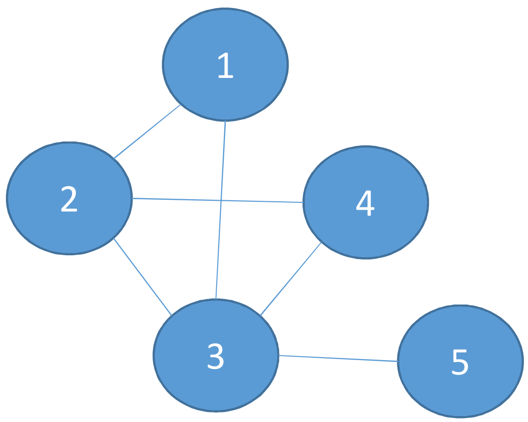
6 2 1 4 3 7 8

Breadth First Search

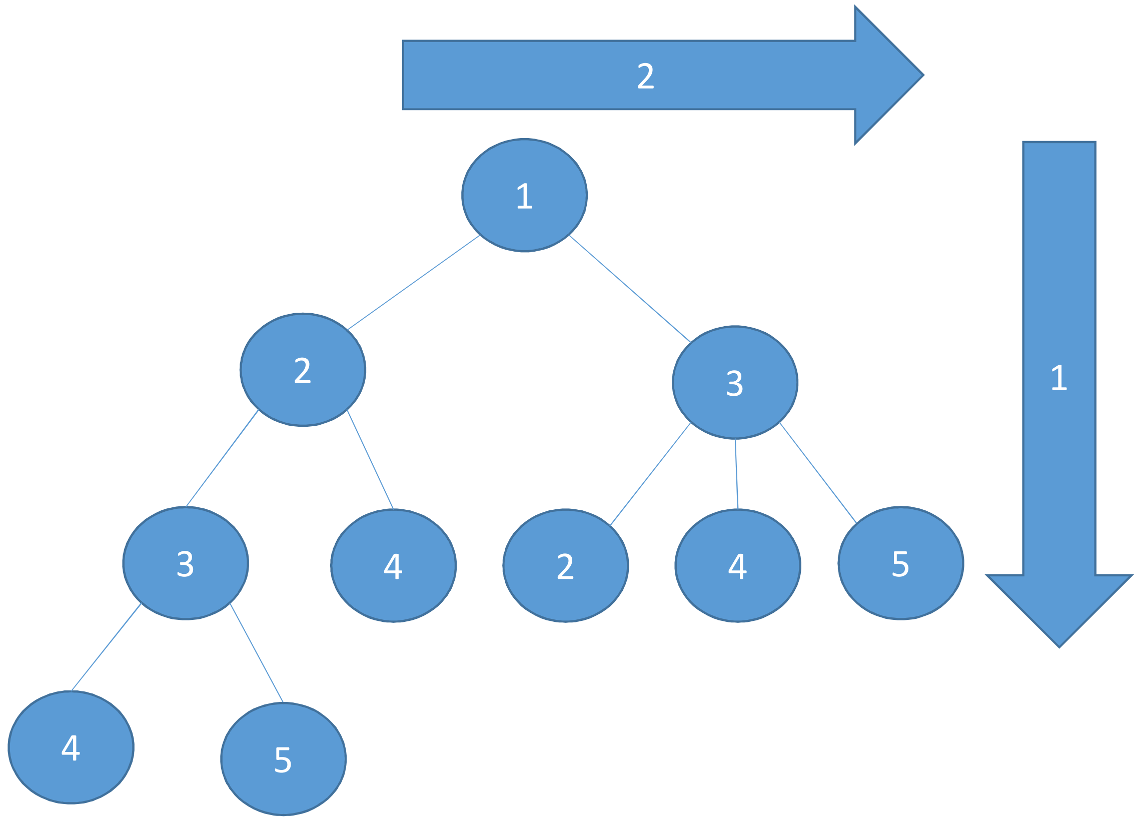


6 2 8 1 4 3 7

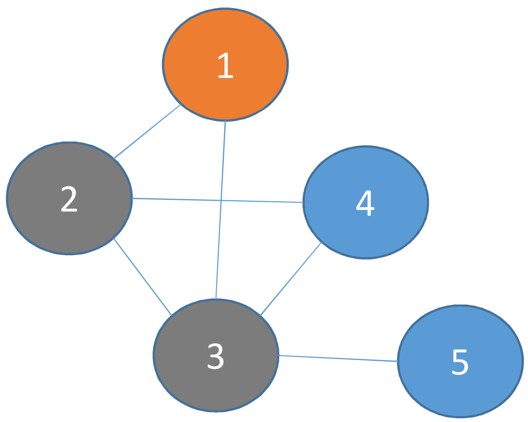
GRAPH



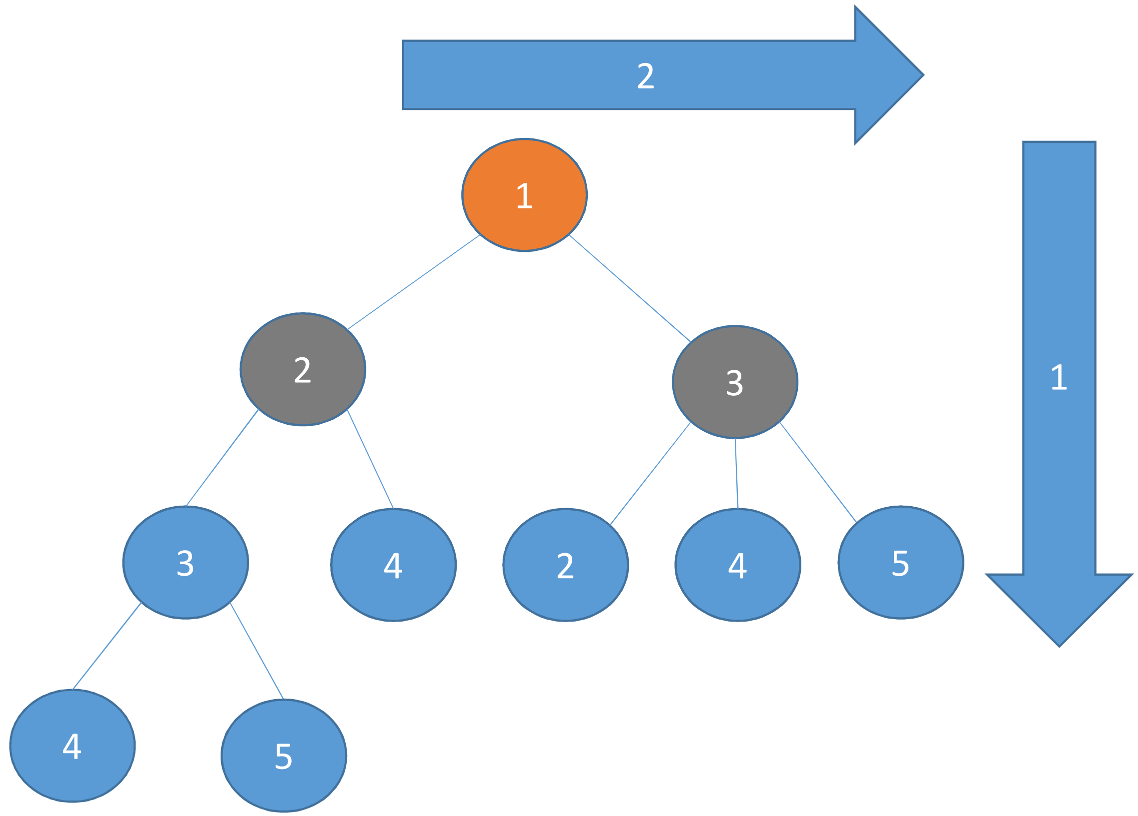
Depth First Search



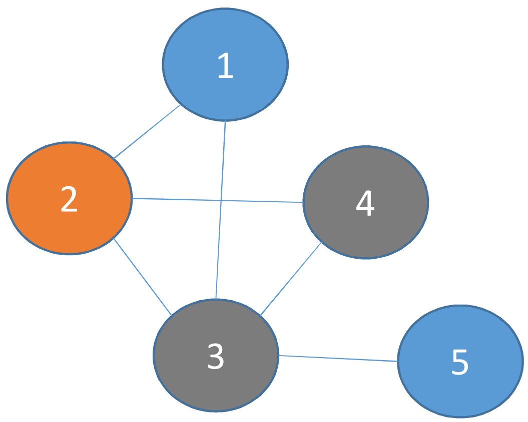
GRAPH



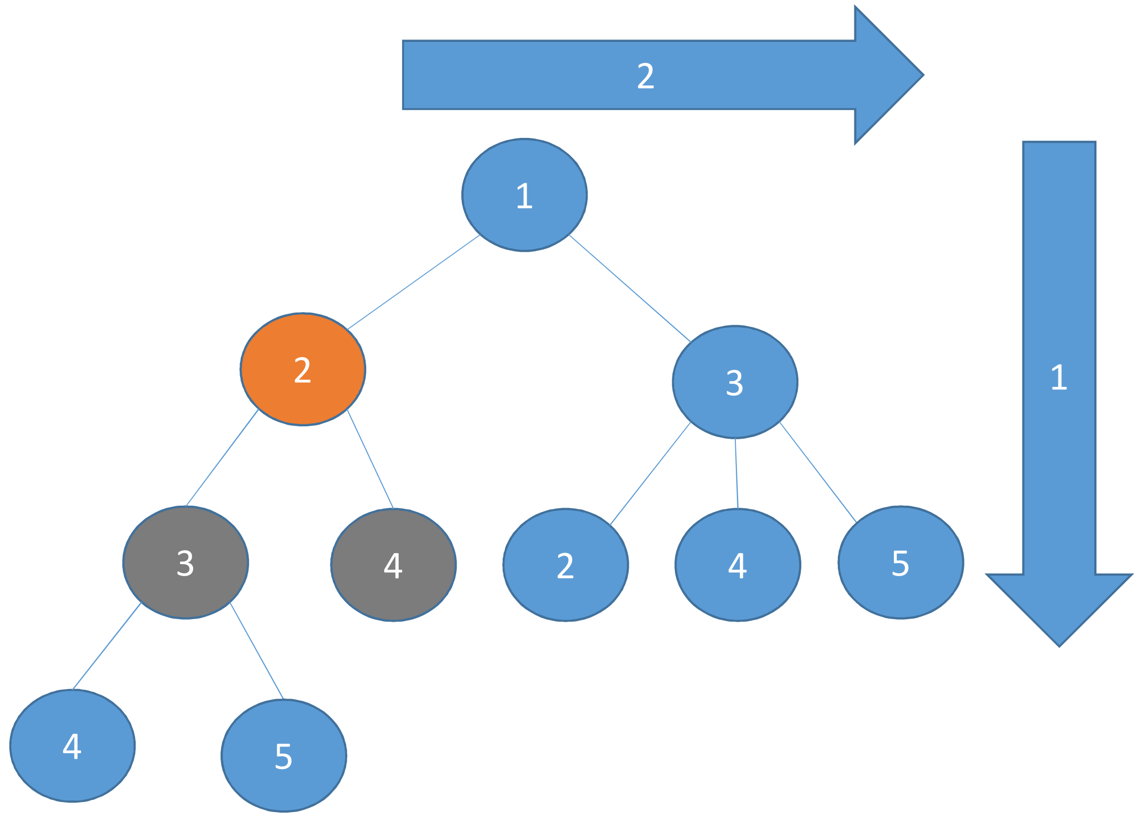
Depth First Search



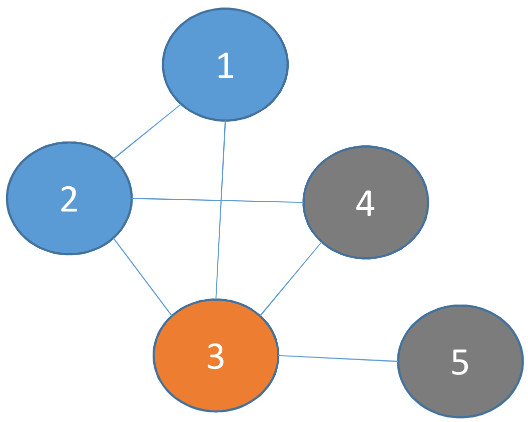
GRAPH



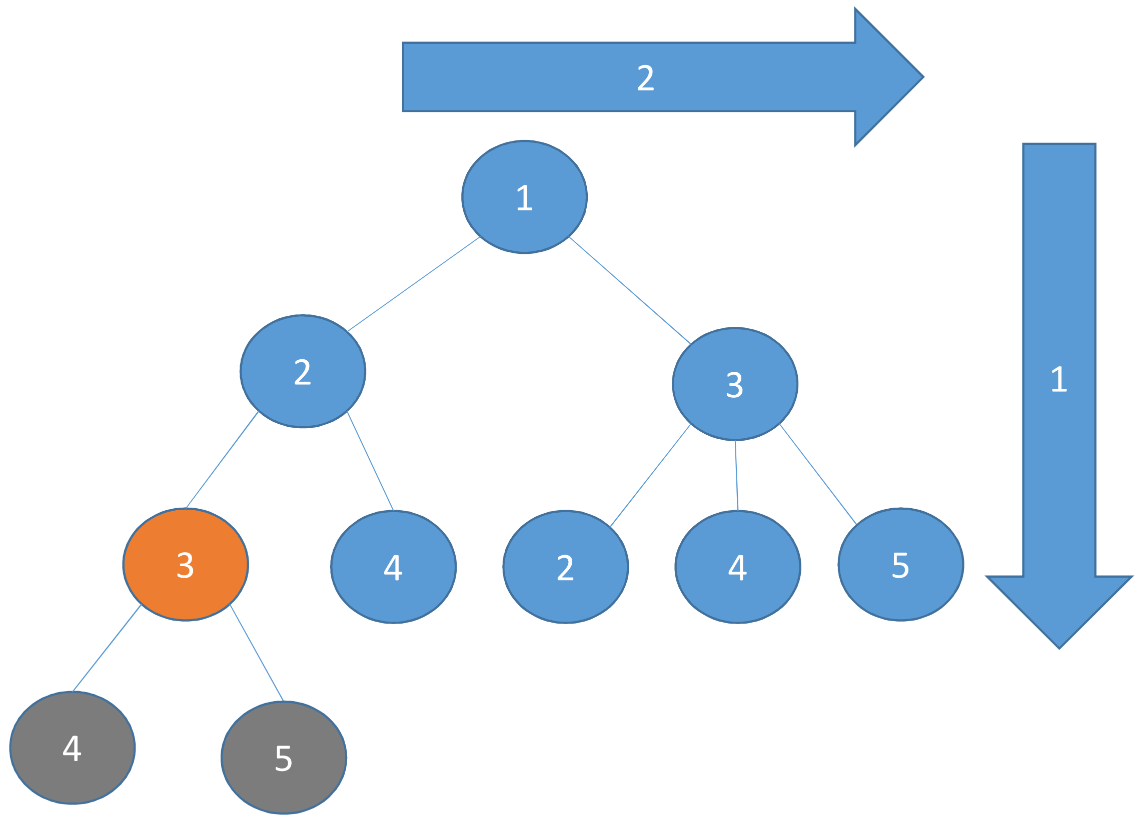
Depth First Search



GRAPH

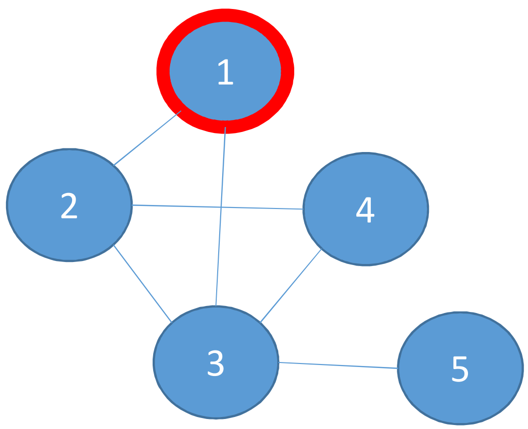


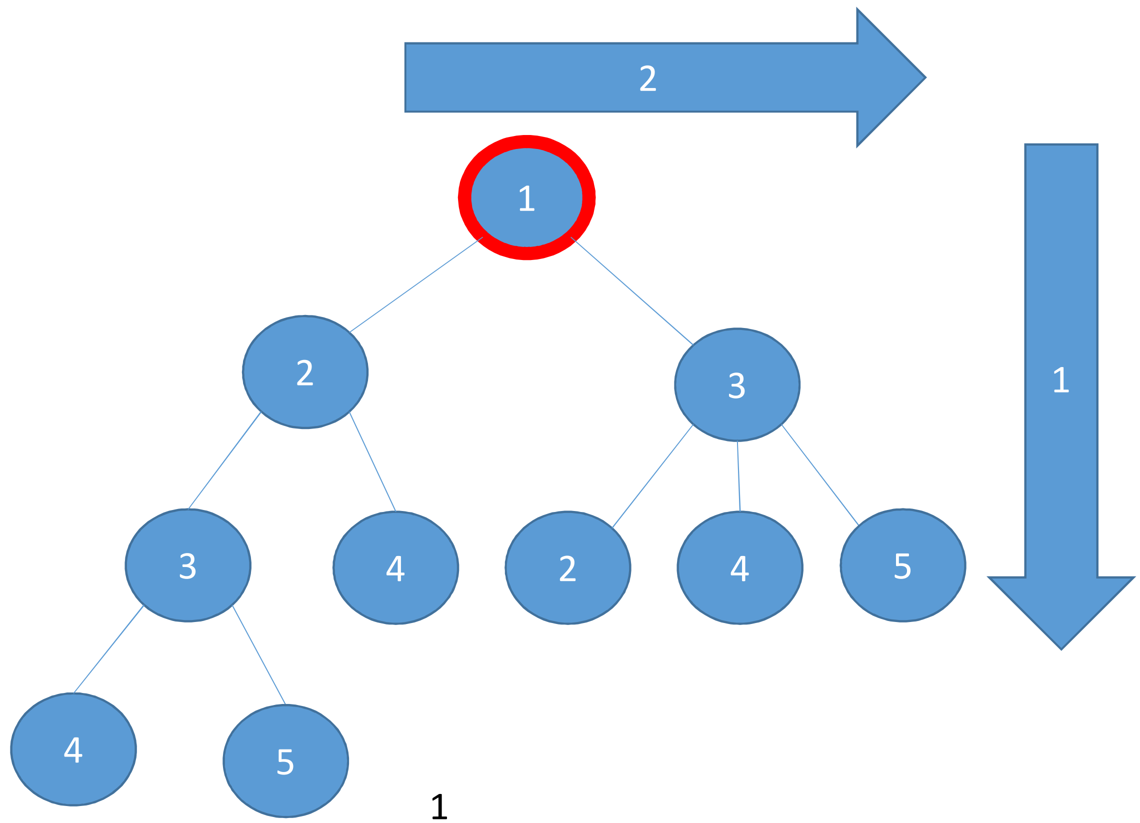
Depth First Search



GRAPH

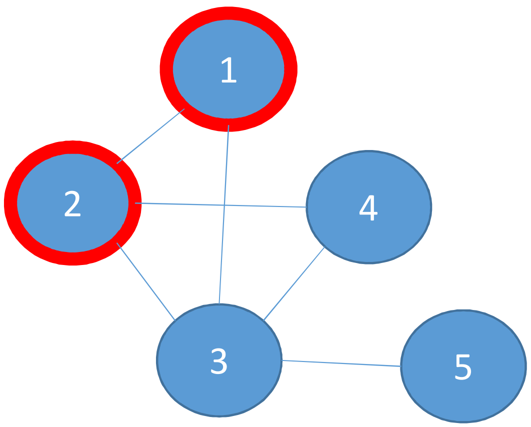
Depth First Search



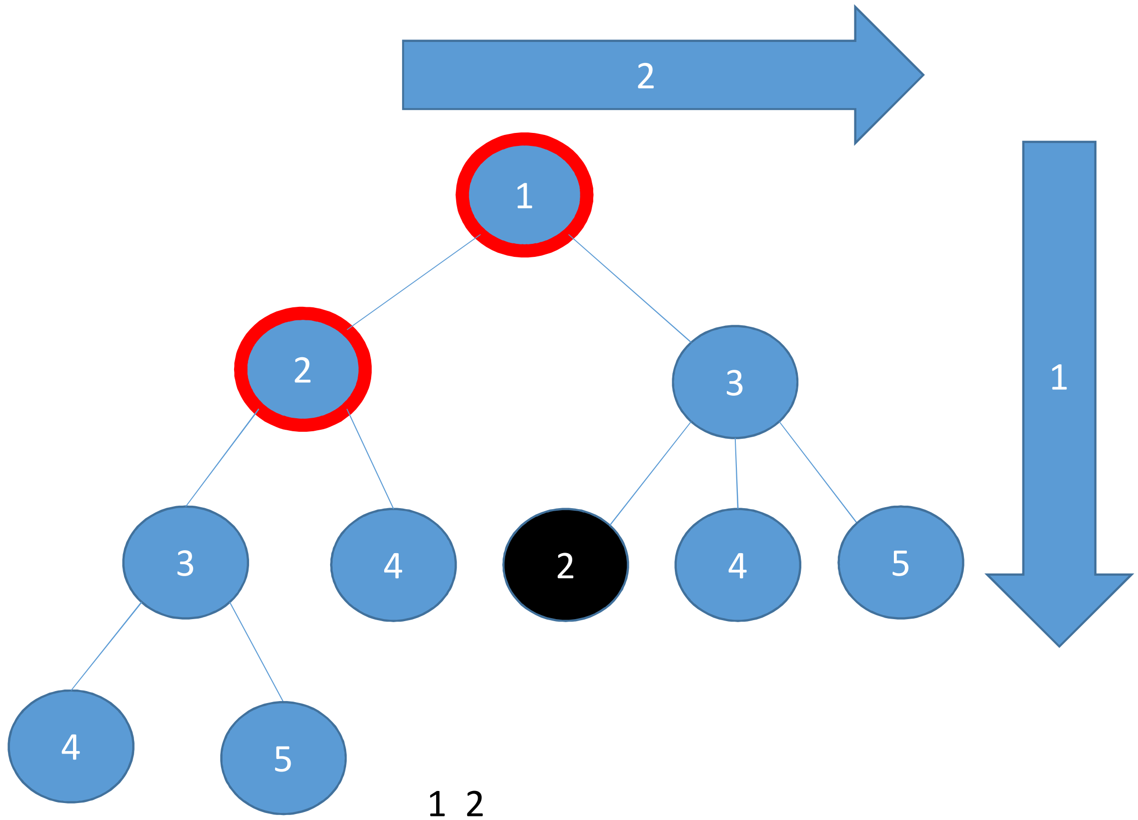


1

GRAPH

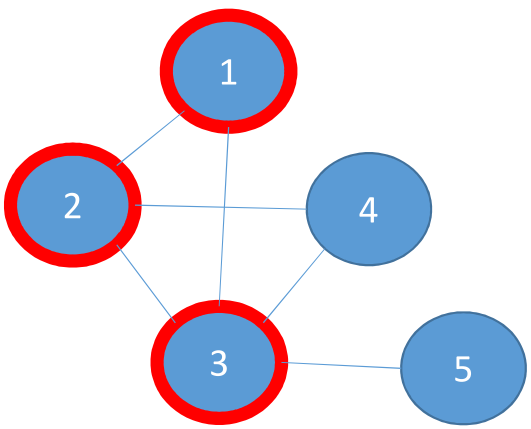


Depth First Search

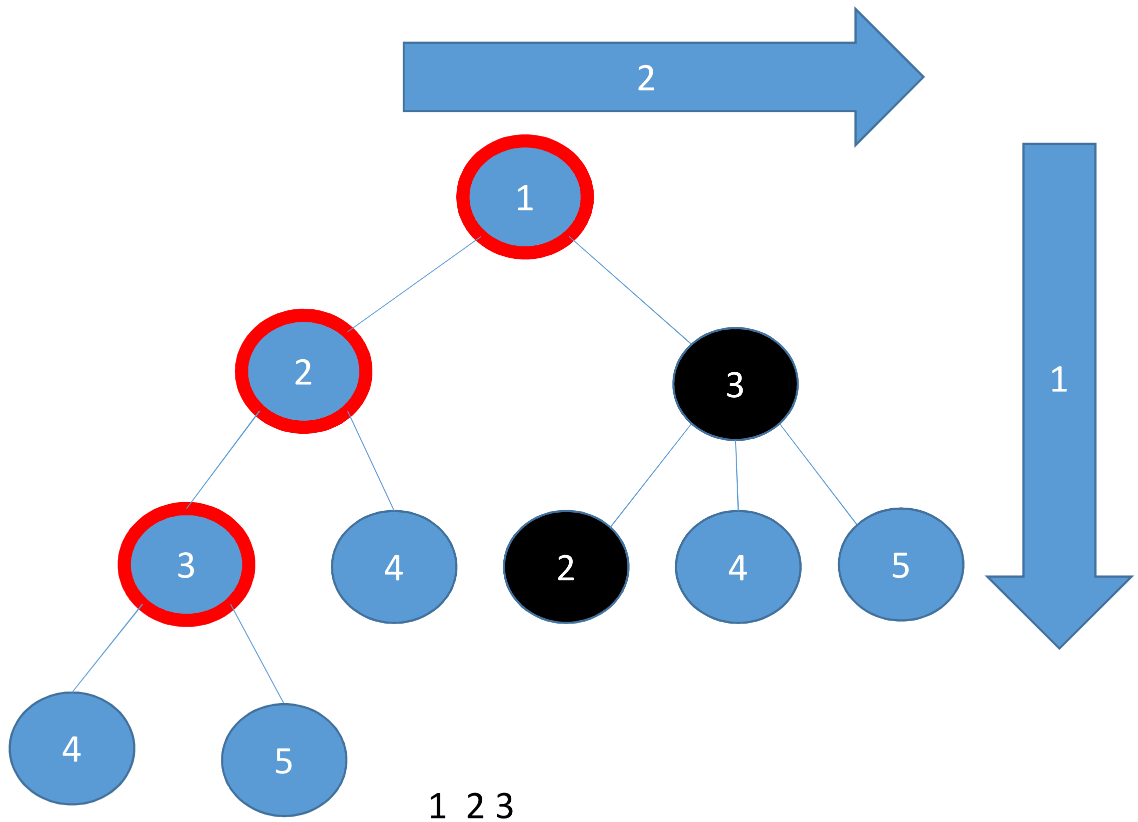


1 2

GRAPH

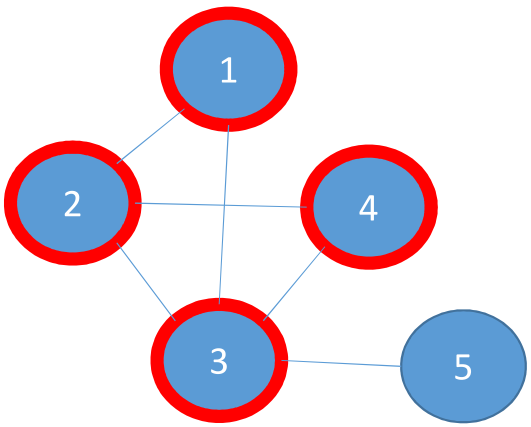


Depth First Search

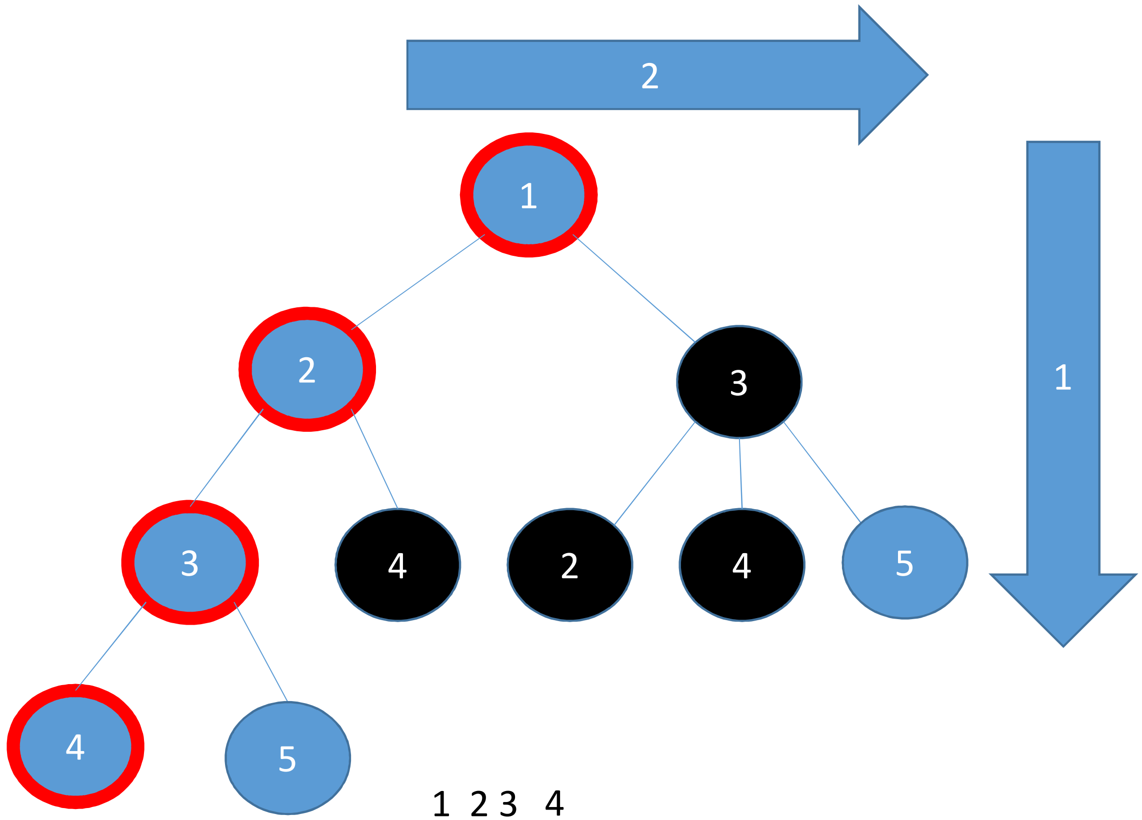


1 2 3

GRAPH



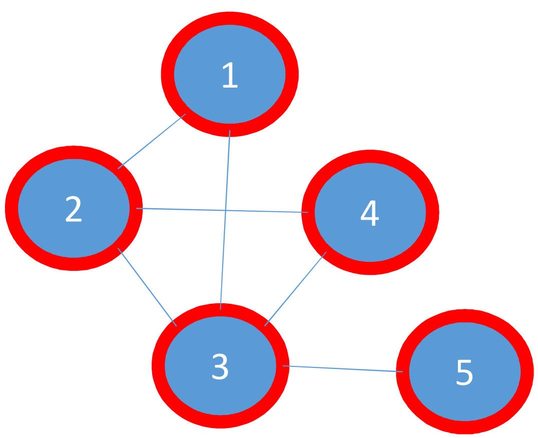
Depth First Search

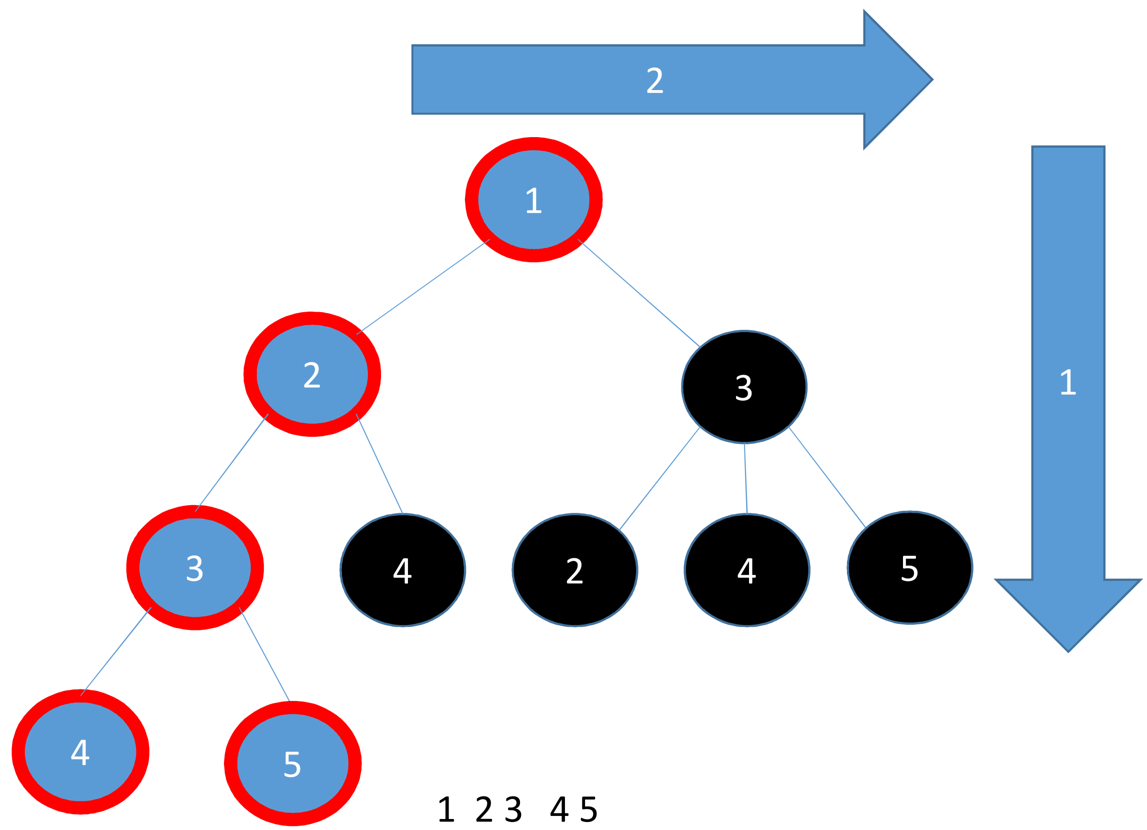


1 2 3 4

GRAPH

Depth First Search

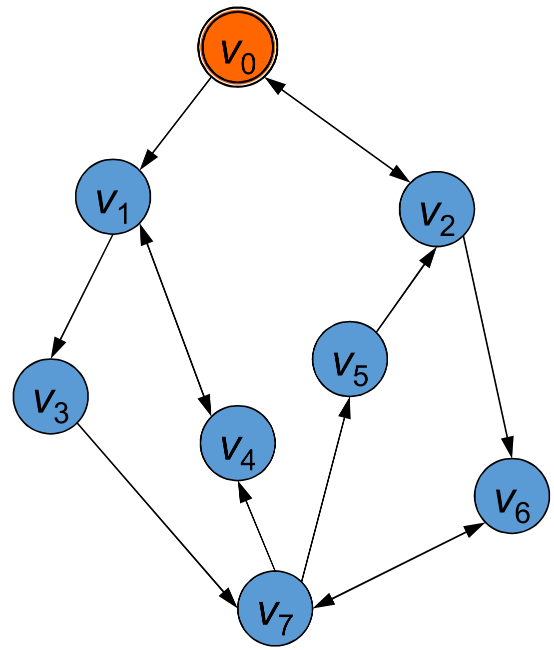




1 2 3 4 5

Depth First Search

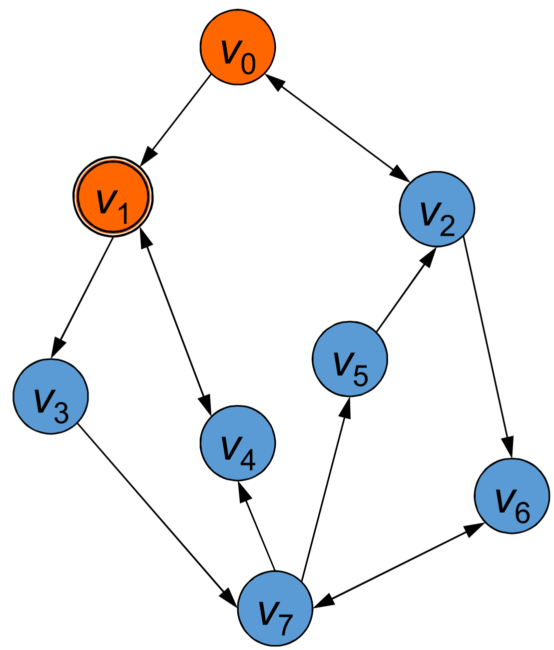
- Start from a vertex v.
- "Visit" vertex v. (i.e., mark v as visited.)
- For each unvisited vertex w adjacent to v
  - Do a DFS starting from vertex w.



Order of visit: v_0

Depth First Search

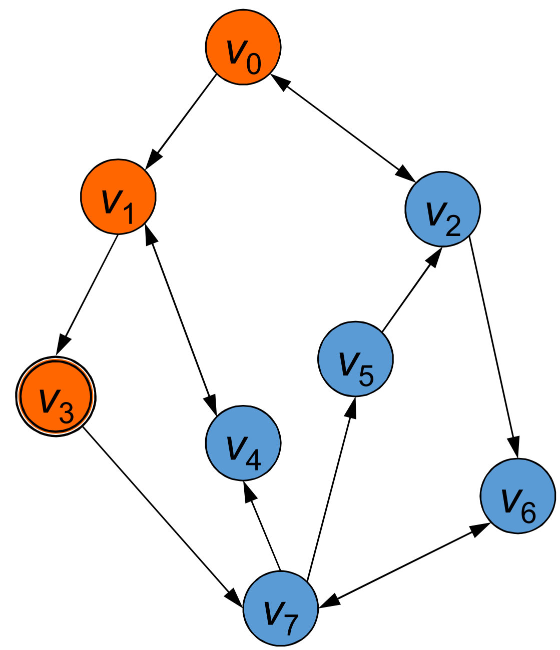
- Start from a vertex v.
- "Visit" vertex v. (i.e., mark v as visited.)
- For each unvisited vertex w adjacent to v
  - Do a DFS starting from vertex w.



Order of visit: v_0, v_1

Depth First Search

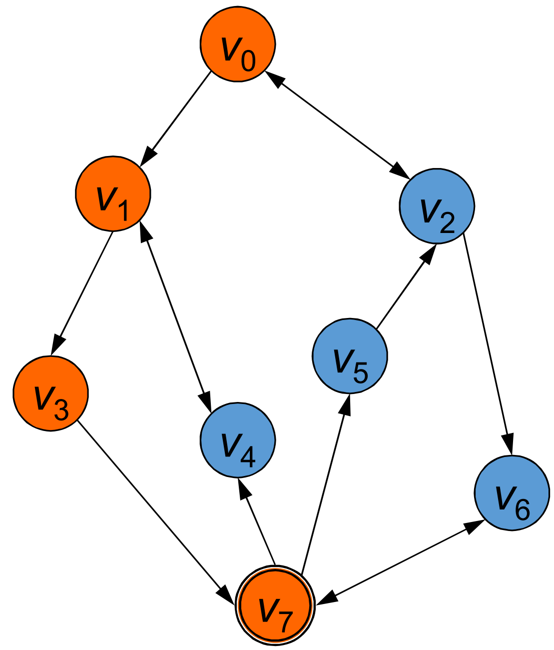
- Start from a vertex *v*.
- "Visit" vertex *v*. (i.e., mark *v* as visited.)
- For each unvisited vertex *w* adjacent to *v*
  - Do a DFS starting from vertex *w*.



Order of visit: v_0, v_1, v_3

Depth First Search

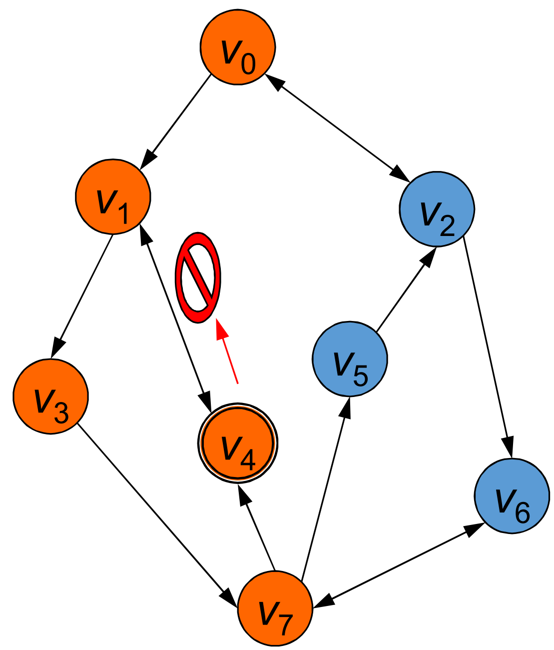
- Start from a vertex *v*.
- "Visit" vertex *v*. (i.e., mark *v* as visited.)
- For each unvisited vertex *w* adjacent to *v*
  - Do a DFS starting from vertex *w*.



Order of visit: v_0, v_1, v_3, v_7

Depth First Search

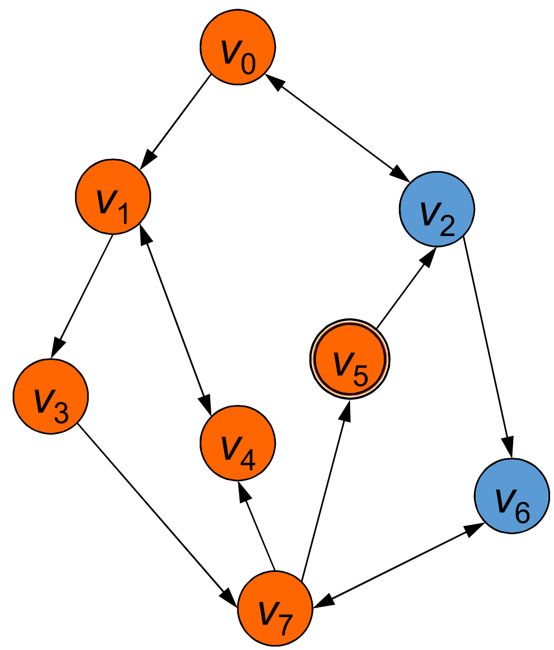
- Start from a vertex *v*.
- “Visit” vertex *v*. (i.e., mark *v* as visited.)
- For each unvisited vertex *w* adjacent to *v*
  - Do a DFS starting from vertex *w*.



Order of visit: v_0, v_1, v_3, v_7, v_4

Depth First Search

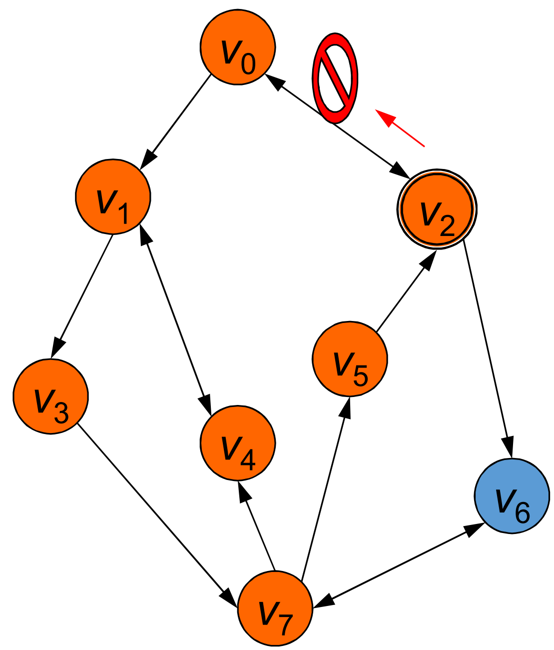
- Start from a vertex *v*.
- "Visit" vertex *v*. (i.e., mark *v* as visited.)
- For each unvisited vertex *w* adjacent to *v*
  - Do a DFS starting from vertex *w*.



Order of visit: v_0, v_1, v_3, v_7, v_4, v_5

Depth First Search

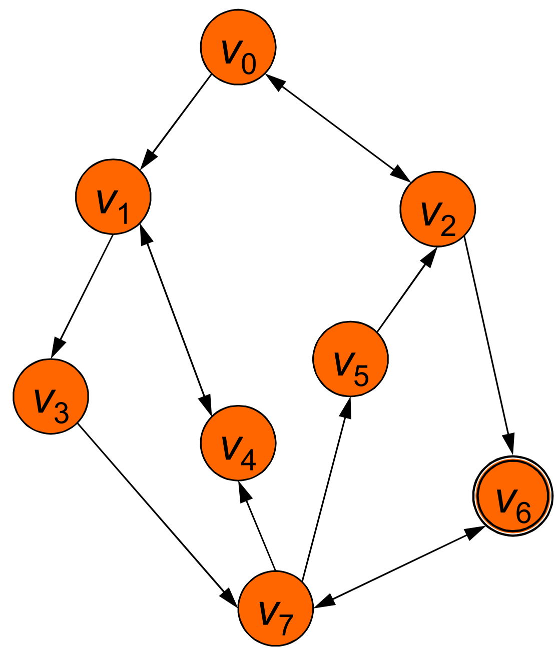
- Start from a vertex v.
- "Visit" vertex v. (i.e., mark v as visited.)
- For each unvisited vertex w adjacent to v
  - Do a DFS starting from vertex w.



Order of visit: v_0, v_1, v_3, v_7, v_4, v_5, v_2

Depth First Search

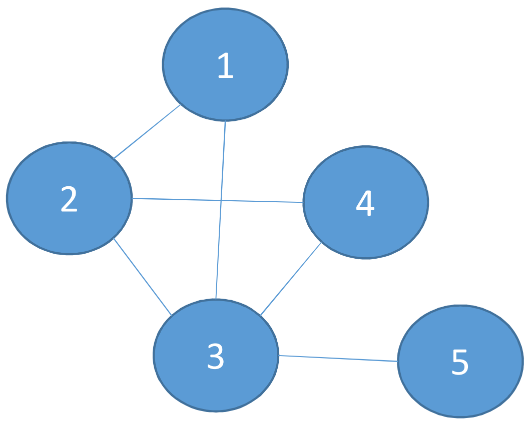
- Start from a vertex v.
- "Visit" vertex v. (i.e., mark v as visited.)
- For each unvisited vertex w adjacent to v
  - Do a DFS starting from vertex w.



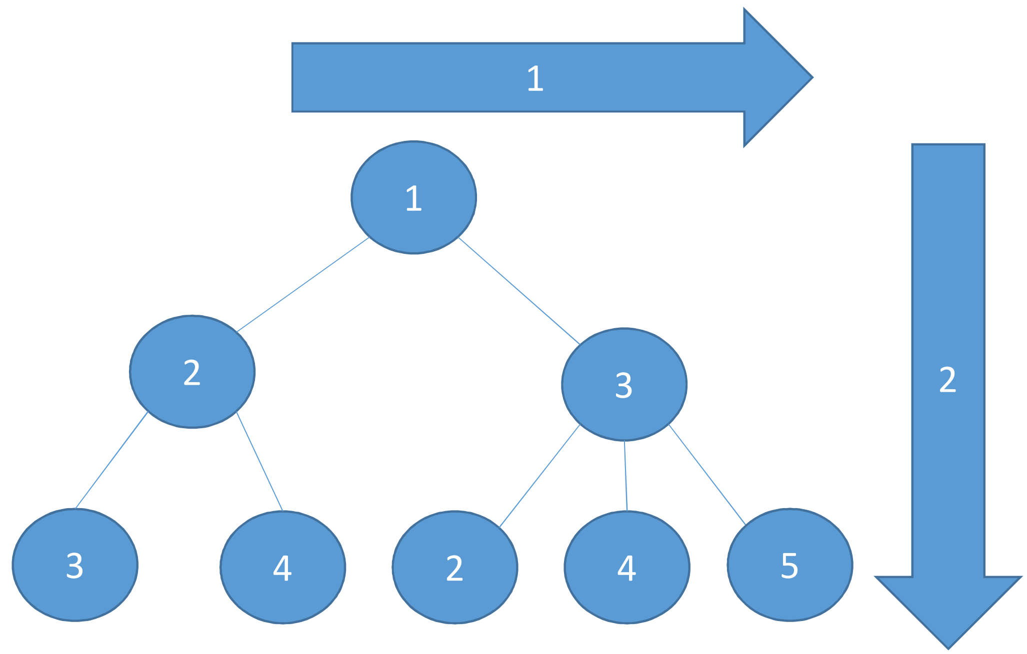
Order of visit: v_0, v_1, v_3, v_7, v_4, v_5, v_2, v_6

DFS is similar to *preorder* tree traversal.

GRAPH

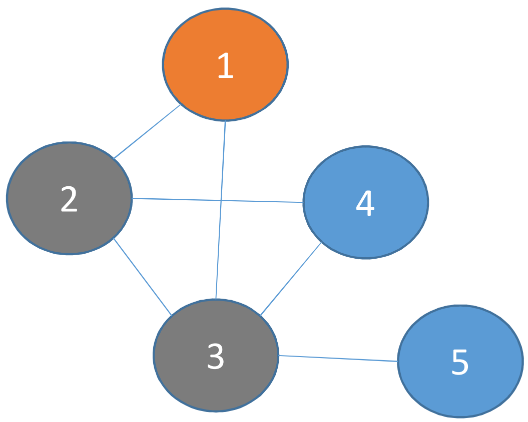


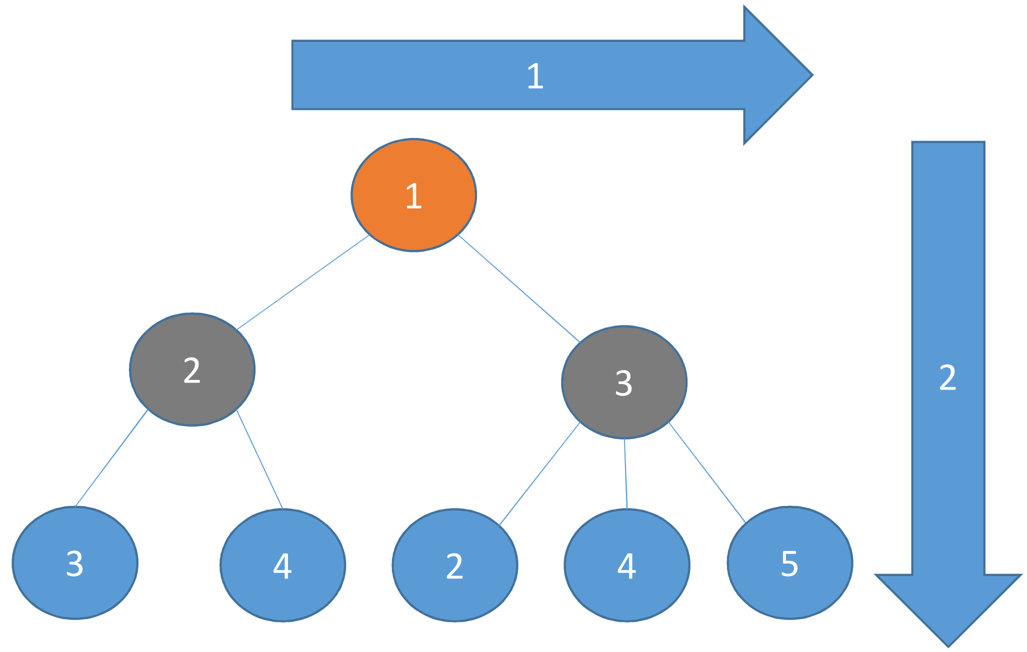
Breadth First Search



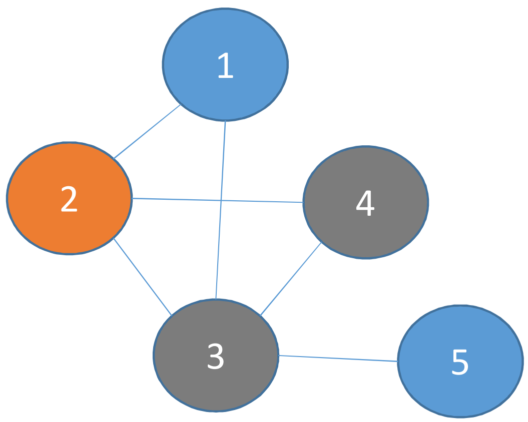
GRAPH

Breadth First Search

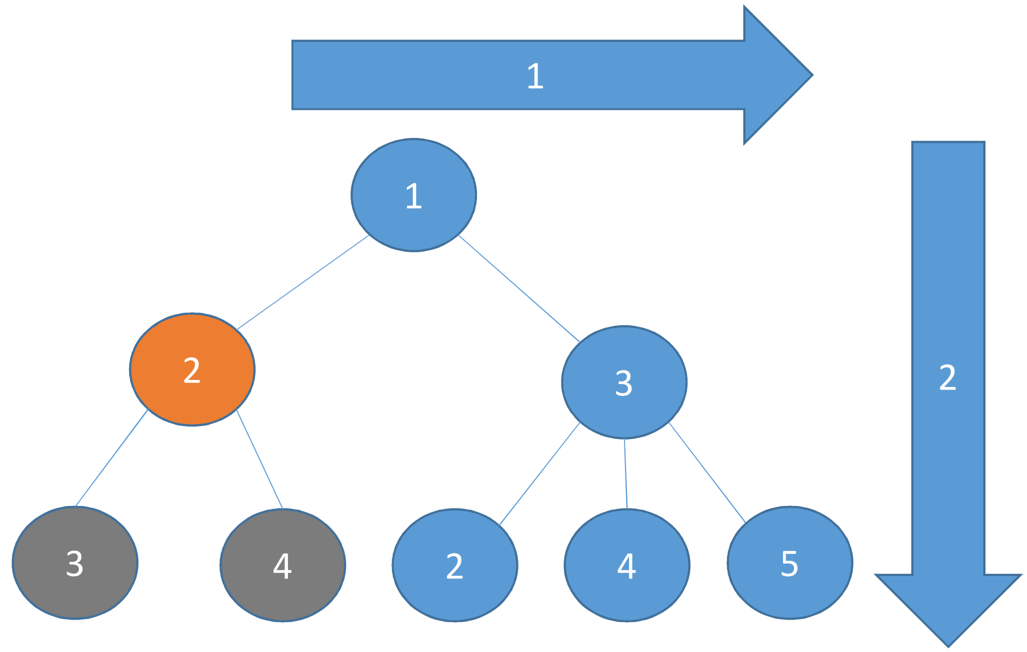




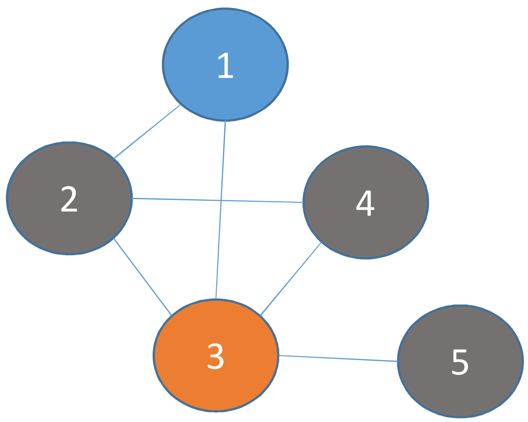
GRAPH



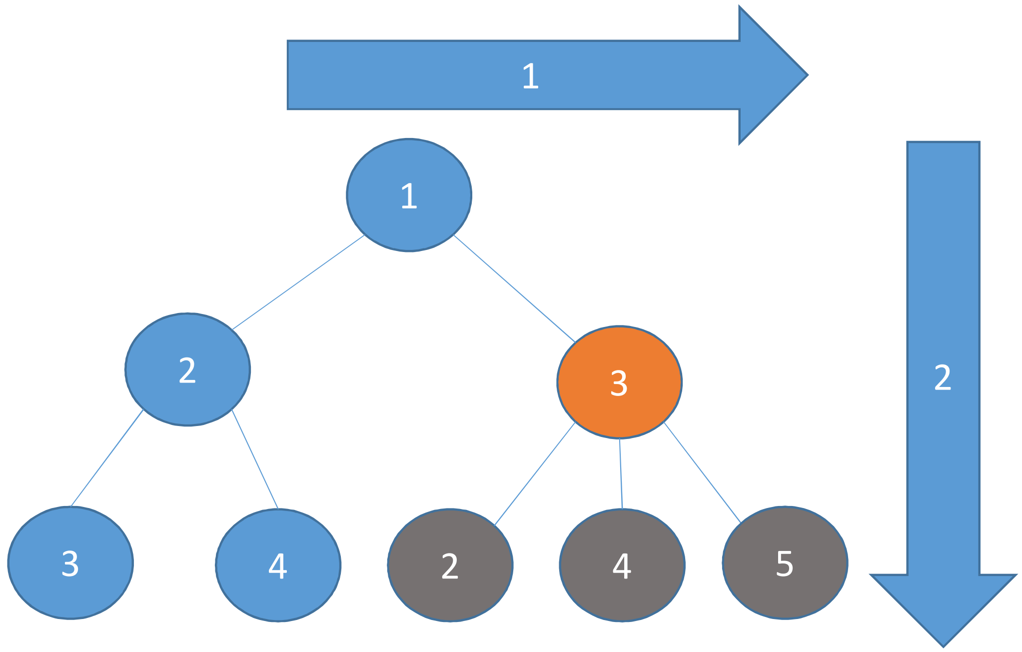
Breadth First Search



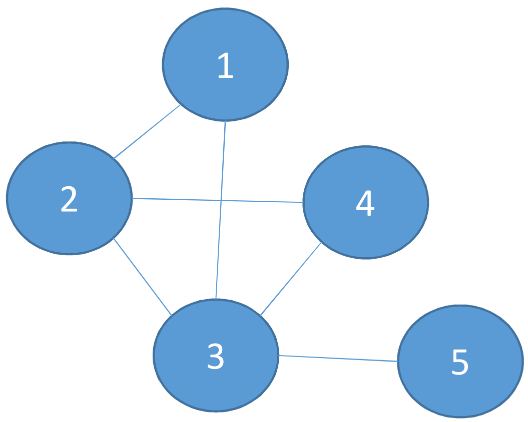
GRAPH



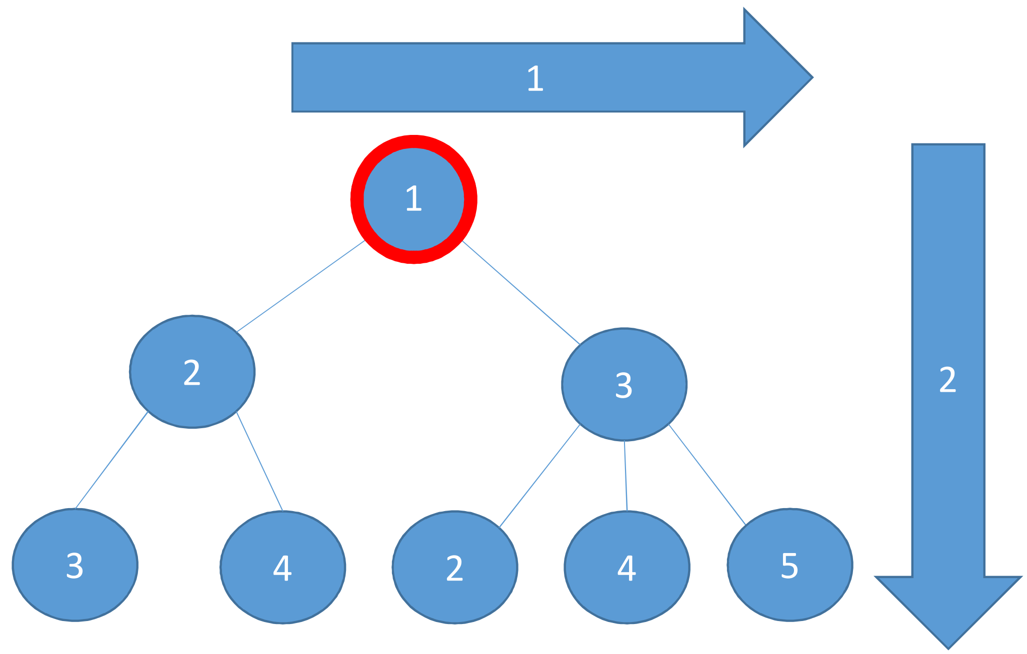
Breadth First Search



GRAPH



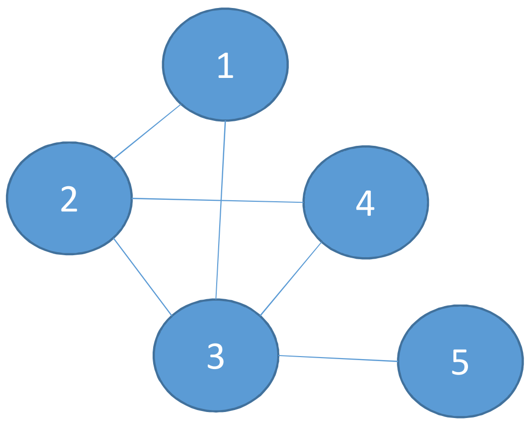
Breadth First Search

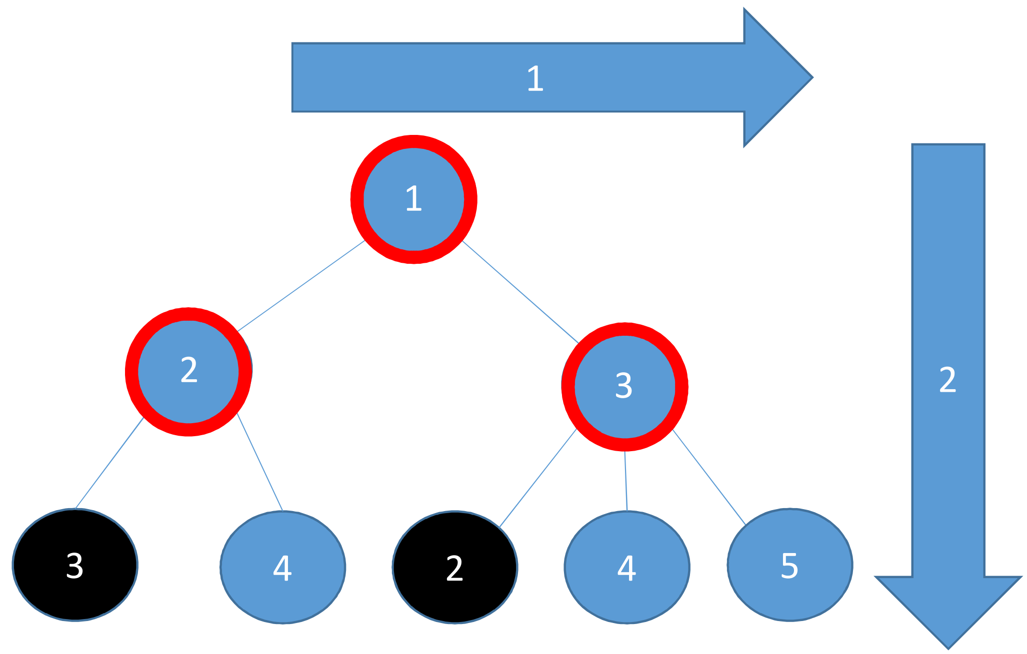


1

GRAPH

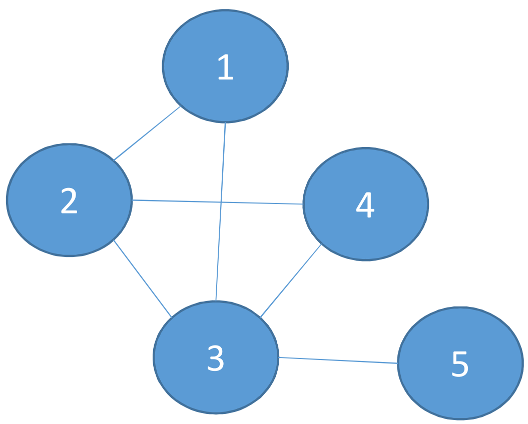
Breadth First Search



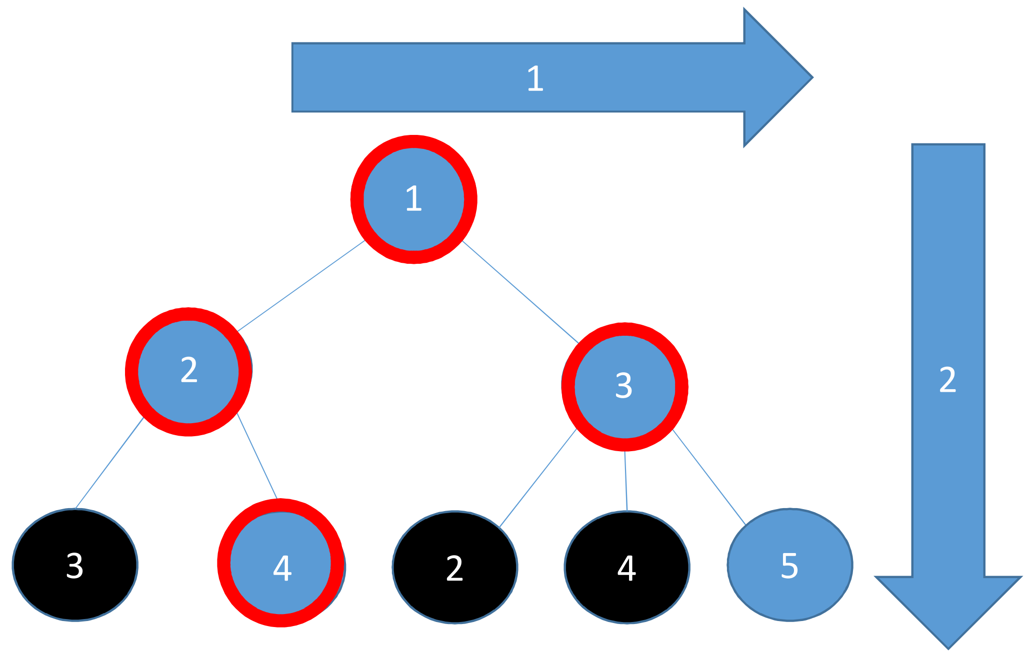


1 2 3

GRAPH

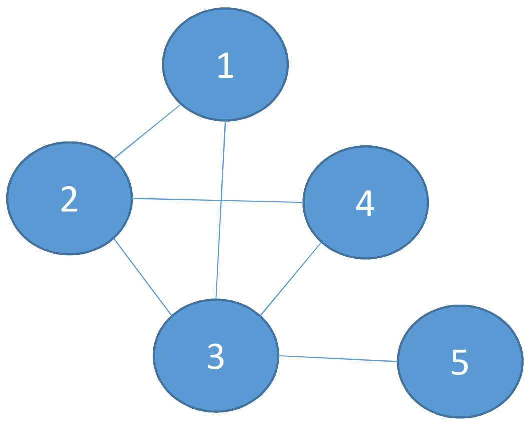


Breadth First Search

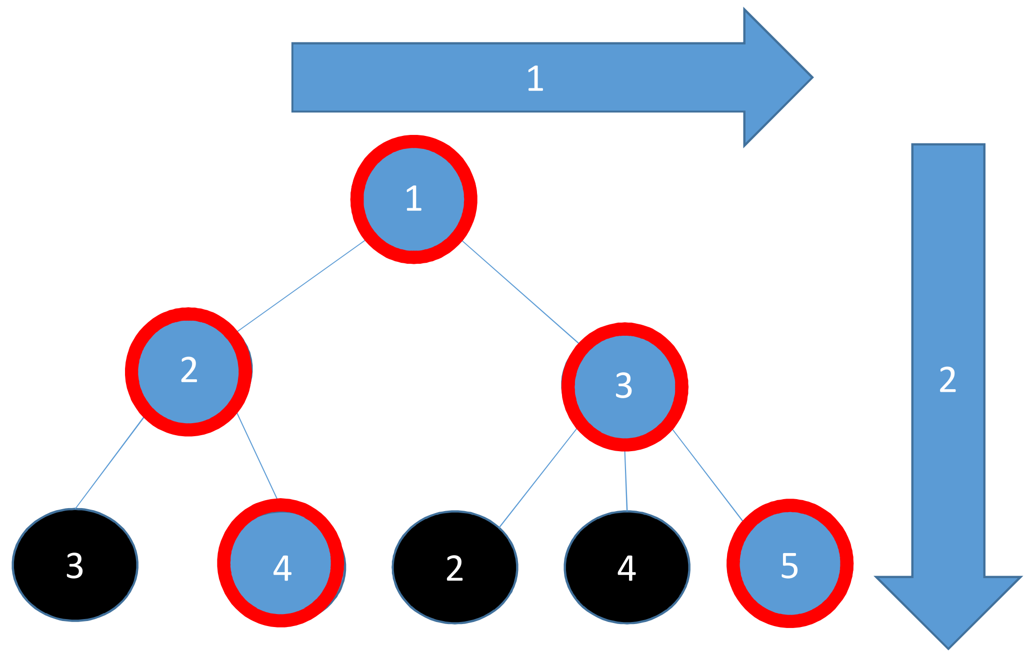


1 2 3 4

GRAPH



Breadth First Search



1 2 3 4 5

Depth First Search

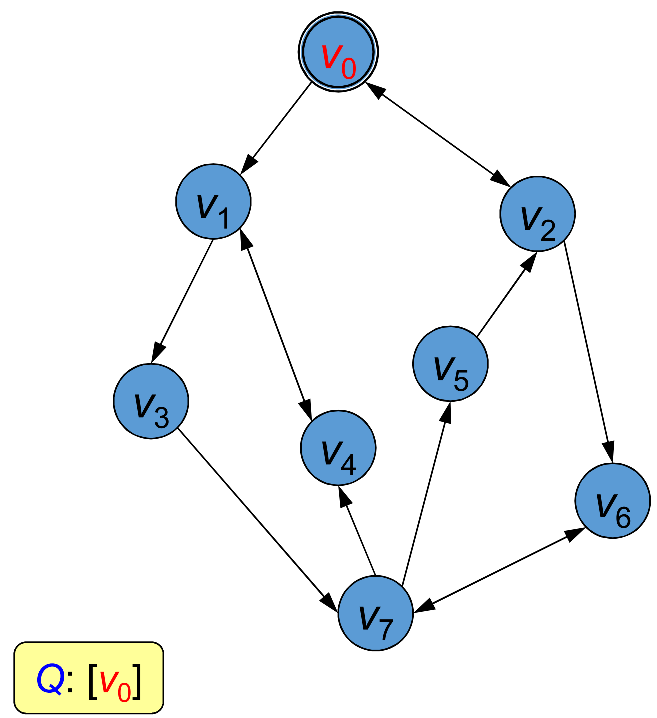
- Start from a vertex *v*.
- “Visit” vertex *v*. (i.e., mark *v* as visited.)
- For each unvisited vertex *w* adjacent to *v*
  - Do a DFS starting from vertex *w*.

Breadth First Search

- Start from a vertex *v*.
- Enqueue *v* to a queue *Q* and mark *v*
- While *Q* is not empty
  - Dequeue a vertex *u* from *Q*.
  - “Visit” *u*.
  - For each un-marked vertex *w* adjacent to *u*
    - Enqueue *w* to *Q* and mark *w*.

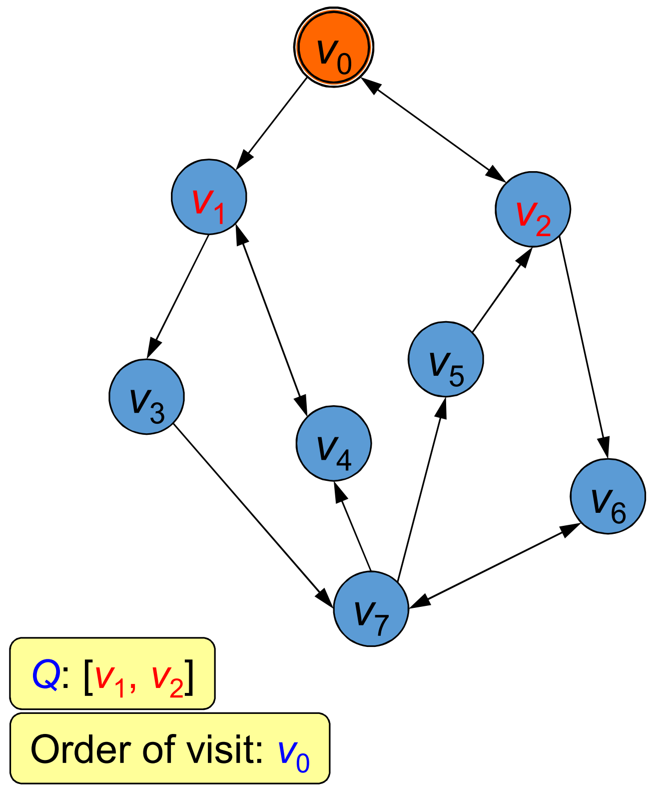
Breadth First Search

- Start from a vertex v.
- Enqueue v to a queue Q and mark v
- While Q is not empty
  - Dequeue a vertex u from Q.
  - "Visit" u.
  - For each un-marked vertex w adjacent to u
    - Enqueue w to Q and mark w.



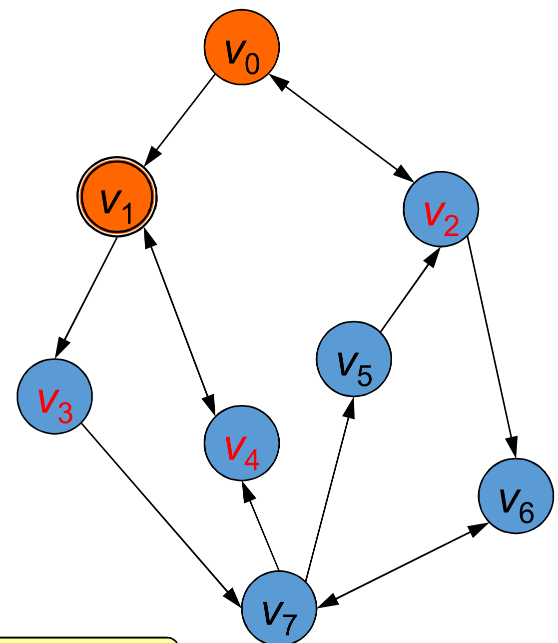
Breadth First Search

- Start from a vertex v.
- Enqueue v to a queue Q and mark v
- While Q is not empty
  - Dequeue a vertex u from Q.
  - "Visit" u.
  - For each un-marked vertex w adjacent to u
    - Enqueue w to Q and mark w.



Breadth First Search

- Start from a vertex v.
- Enqueue v to a queue Q and mark v
- While Q is not empty
  - Dequeue a vertex u from Q.
  - "Visit" u.
  - For each un-marked vertex w adjacent to u
    - Enqueue w to Q and mark w.

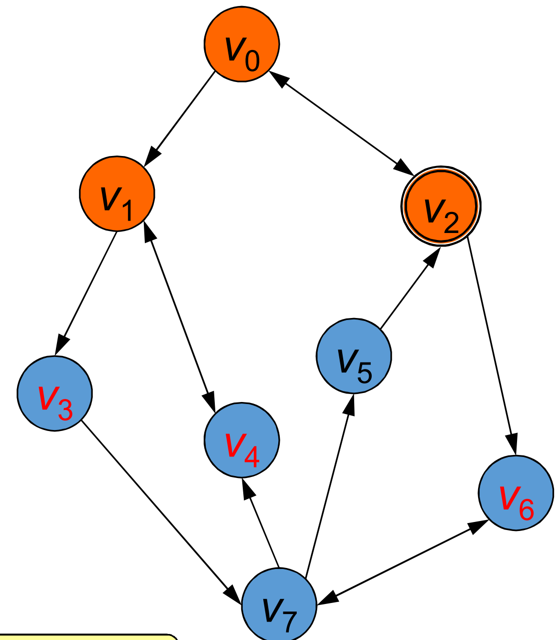


Q: [v_2, v_3, v_4]

Order of visit: v_0, v_1

Breadth First Search

- Start from a vertex *v*.
- Enqueue *v* to a queue *Q* and mark *v*
- While *Q* is not empty
  - Dequeue a vertex *u* from *Q*.
  - “Visit” *u*.
  - For each un-marked vertex *w* adjacent to *u*
    - Enqueue *w* to *Q* and mark *w*.

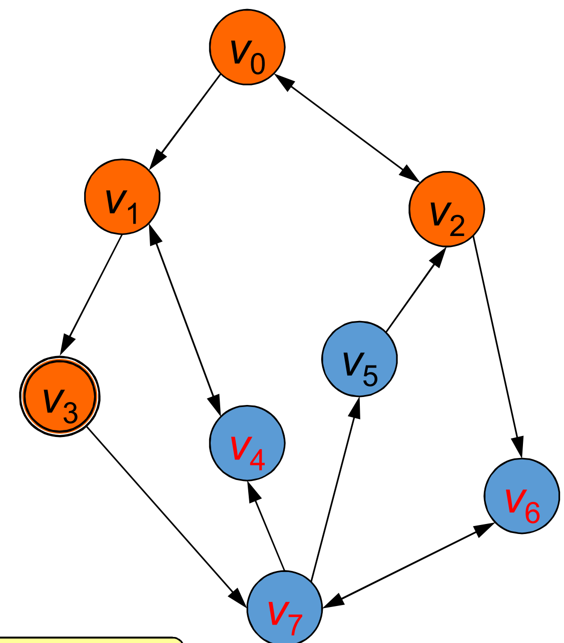


Q: [v_3, v_4, v_6]

Order of visit: v_0, v_1, v_2

Breadth First Search

- Start from a vertex v.
- Enqueue v to a queue Q and mark v
- While Q is not empty
  - Dequeue a vertex u from Q.
  - "Visit" u.
  - For each un-marked vertex w adjacent to u
    - Enqueue w to Q and mark w.

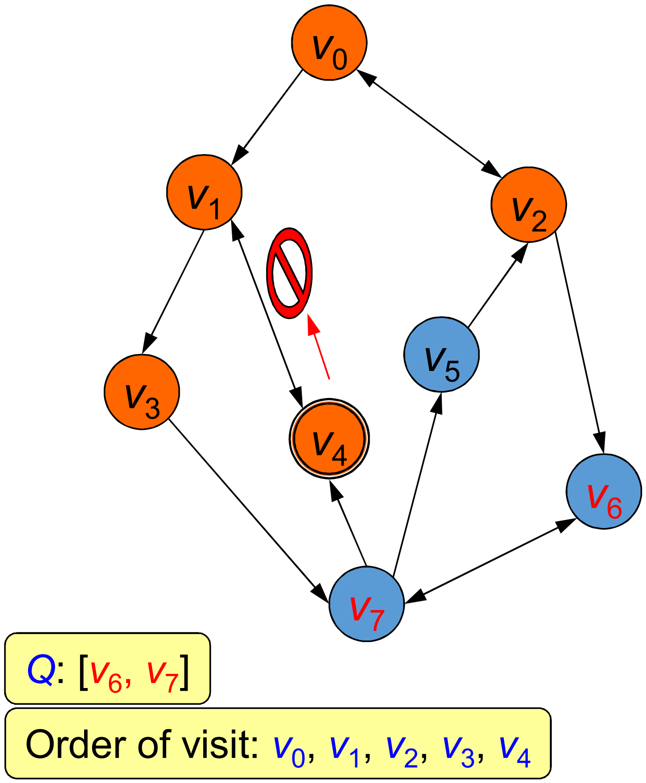


Q: [v_4, v_6, v_7]

Order of visit: v_0, v_1, v_2, v_3

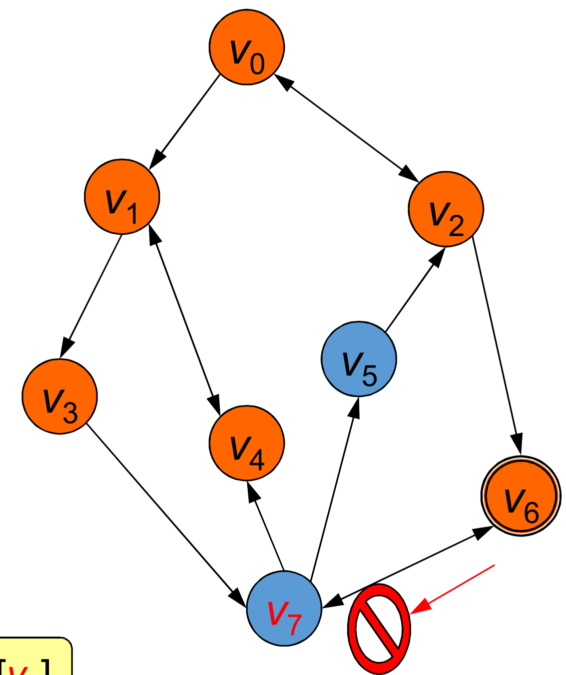
Breadth First Search

- Start from a vertex v.
- Enqueue v to a queue Q and mark v
- While Q is not empty
  - Dequeue a vertex u from Q.
  - "Visit" u.
  - For each un-marked vertex w adjacent to u
    - Enqueue w to Q and mark w.



Breadth First Search

- Start from a vertex v.
- Enqueue v to a queue Q and mark v
- While Q is not empty
  - Dequeue a vertex u from Q.
  - "Visit" u.
  - For each un-marked vertex w adjacent to u
    - Enqueue w to Q and mark w.

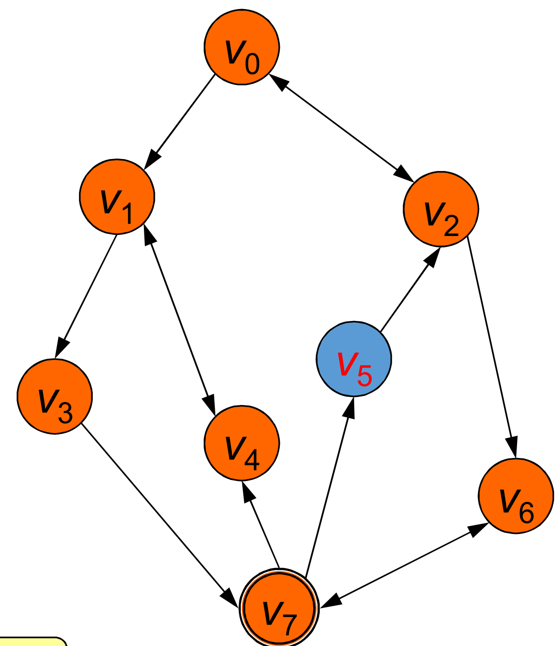


Q: [v_7]

Order of visit: v_0, v_1, v_2, v_3, v_4, v_6

Breadth First Search

- Start from a vertex v.
- Enqueue v to a queue Q and mark v
- While Q is not empty
  - Dequeue a vertex u from Q.
  - "Visit" u.
  - For each un-marked vertex w adjacent to u
    - Enqueue w to Q and mark w.

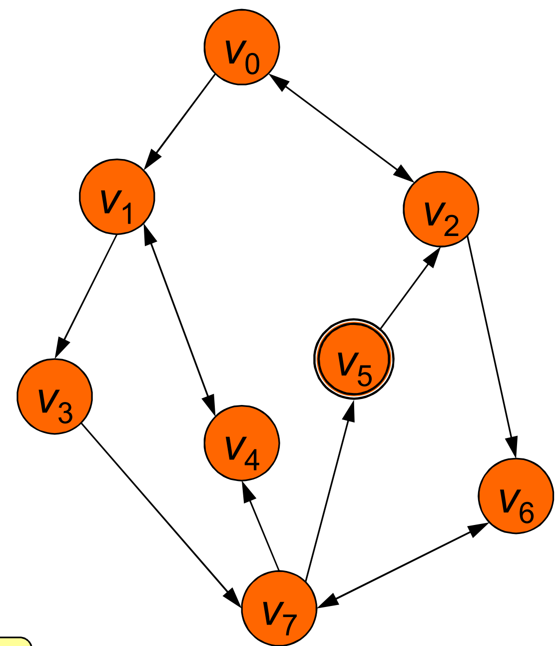


Q: [v_5]

Order of visit: v_0, v_1, v_2, v_3, v_4, v_6, v_7

Breadth First Search

- Start from a vertex *v*.
- Enqueue *v* to a queue *Q* and mark *v*
- While *Q* is not empty
  - Dequeue a vertex *u* from *Q*.
  - "Visit" *u*.
  - For each un-marked vertex *w* adjacent to *u*
    - Enqueue *w* to *Q* and mark *w*.



Q: []

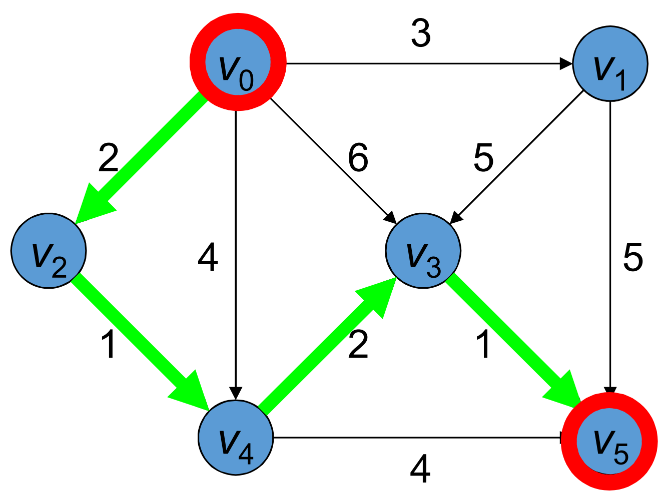
Order of visit: v_0, v_1, v_2, v_3, v_4, v_6, v_7, v_5

Graph algorithms

- **Graph Traversal (Graph Searching)**
  - Breadth-first search
  - Depth-first search
- **Shortest-Path Algorithm**
  - Dijkstra’s algorithm
- **Minimum Spanning Tree**
  - Prim’s Algortihm
  - Kruskal’s Algorithm

Shortest Path Problem

- To find the path between two vertices such that the sum of the weights in the path is minimized.



other paths and lengths

$v_0\ v_1\ v_5$: 3 + 5 = 8

$v_0\ v_1\ v_3\ v_5$: 3 + 5 + 1 = 9

$v_0\ v_3\ v_5$: 6 + 1 = 7

$v_0\ v_4\ v_5$: 4 + 4 = 8

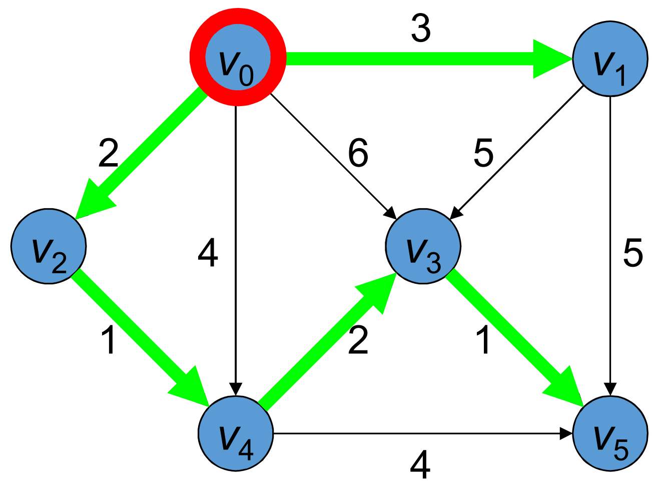
$v_0\ v_2\ v_4\ v_5$: 2 + 1 + 4 = 7

Edge in a shortest path

Shortest path from v_0 to v_5 : $v_0\ v_2\ v_4\ v_3\ v_5$
Path length : 2 + 1 + 2 + 1 = 6

Shortest Path Problem

- To find the path between two vertices such that the sum of the weights in the path is minimized.



Shortest paths from v_0 to other vertices

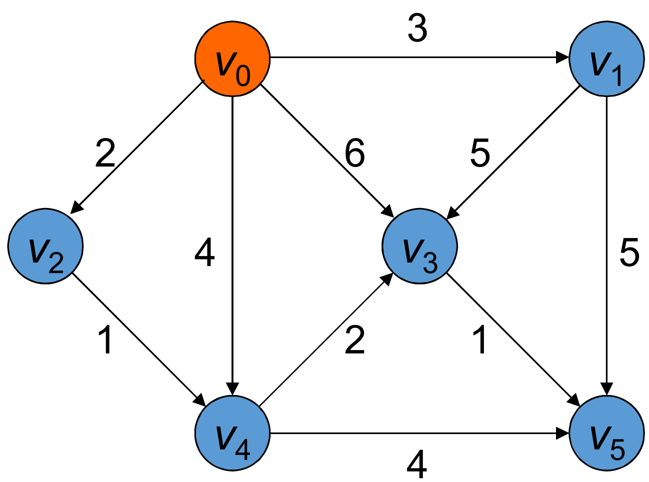
<u>shortest paths</u>

v_0 to v_1: v_0, v_1
v_0 to v_2: v_0, v_2
v_0 to v_3: v_0, v_2, v_4, v_3
v_0 to v_4: v_0, v_2, v_4
v_0 to v_5: v_0, v_2, v_4, v_3, v_5

Edge in a shortest path

Dijkstra’s Algorithm

1. Initialize the cost/distance table



Current best cost from v_0

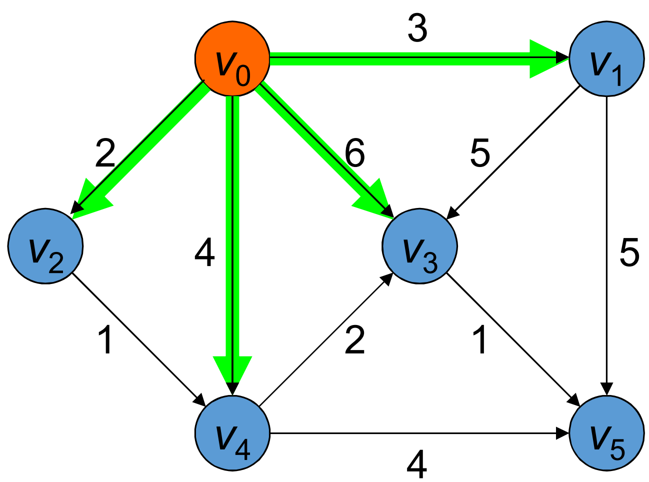
vertex	distance	previous
v_0	0	0
v_1	∞	0
v_2	∞	0
v_3	∞	0
v_4	∞	0
v_5	∞	0

Dijkstra's Algorithm

1. Initialize the cost/distance table
2. Pick the unvisited vertex with the min cost and mark it as visited.
3. Update the best cost of the adjacent vertices if needed.

v_0

Order of vertex visited:
v_0



visited vertex

edge in current best path

Current best cost from v_0

vertex	distance	previous
v_0	0	0
v_1	∞	0
v_2	∞	0
v_3	∞	0
v_4	∞	0
v_5	∞	0

$V_0 \rightarrow V_1 = 3$

$V_0 \rightarrow V_2 = 2$

$V_0 \rightarrow V_3 = 6$

$V_0 \rightarrow V_4 = 4$

V_5 is not adjacent to V_0

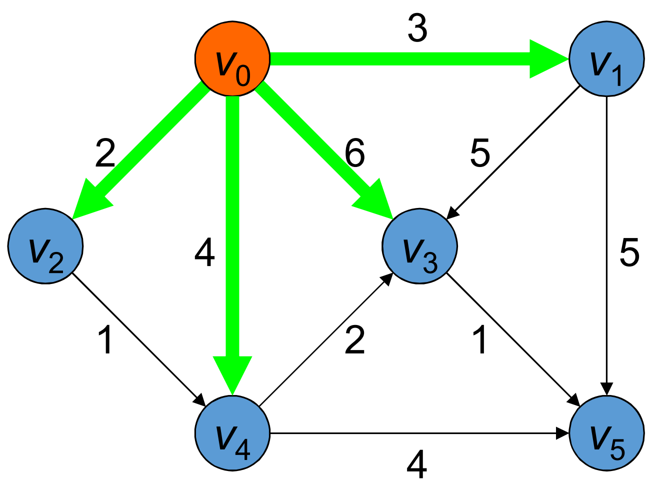
During the algorithm, a visited vertex v_j means the shortest path from v_0 (start) to v_j has been found already.

Repeat if there are unvisited vertices

2. Pick the unvisited vertex with the min cost and mark it as visited.

Dijkstra's Algorithm

Order of vertex visited:
v_0



Current best cost from v_0

vertex	distance	previous
v_0	0	0
v_1	3	v_0
v_2	2	v_0
v_3	6	v_0
v_4	4	v_0
v_5	∞	0

visited vertex

edge in current best path

During the algorithm, a visited vertex v_j means the shortest path from v_0 (start) to v_j has been found already.

Dijkstra's Algorithm

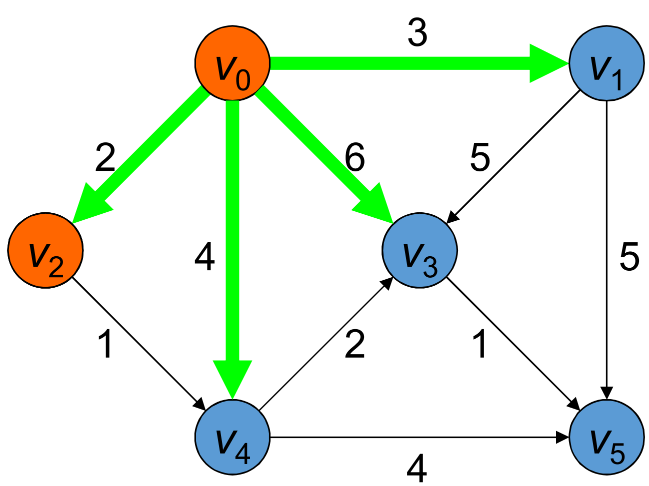
Repeat if there are unvisited vertices

2. Pick the unvisited vertex with the min cost and mark it as visited.

v_2

3. Update the best cost of the adjacent vertices if needed.

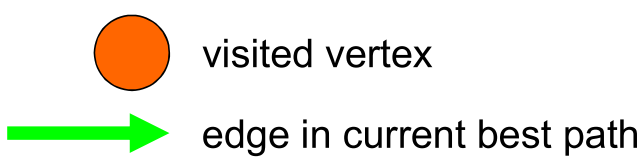
Order of vertex visited:
v_0, v_2



Current best cost from v_0

vertex	distance	previous
v_0	0	0
v_1	3	v_0
v_2	$\boxed{2}$	v_0
v_3	6	v_0
v_4	4	v_0
v_5	∞	0

v_2 has the min cost of 2



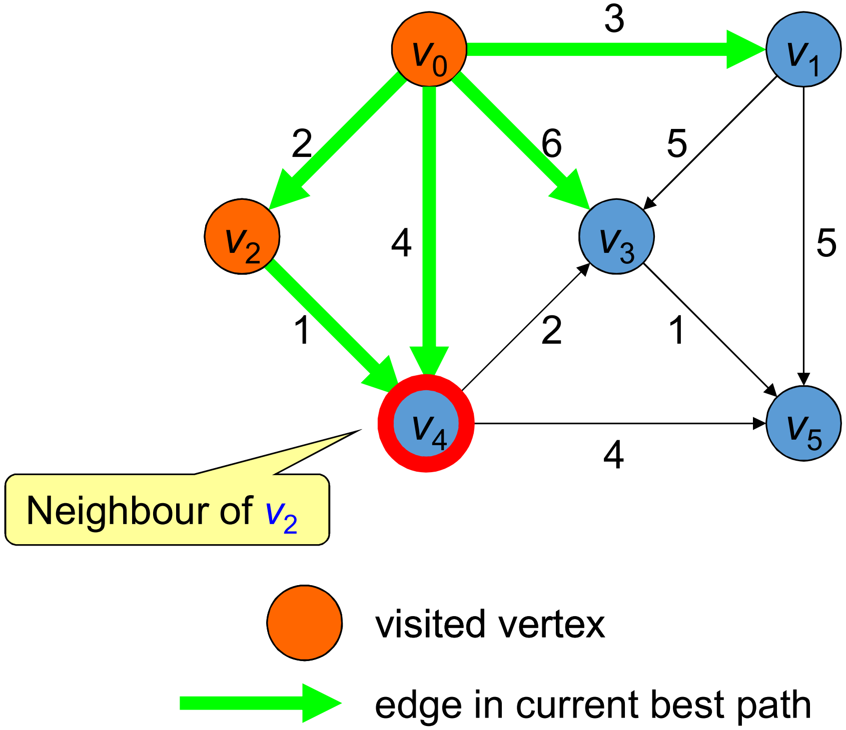
Dijkstra’s Algorithm

Repeat if there are unvisited vertices

2. Pick the unvisited vertex with the min cost and mark it as visited.
3. Update the best cost of the adjacent vertices if needed.

v_2

Order of vertex visited:
v_0, v_2



Current best cost from v_0

vertex	distance	previous
v_0	0	0
v_1	3	v_0
v_2	2	v_0
v_3	6	v_0
v_4	4	v_0
v_5	∞	0

Pick the unvisited vertex with minimum current distance. → v_2
Check the adjacent vertices [v_4] if there are better paths.

Dijkstra’s Algorithm

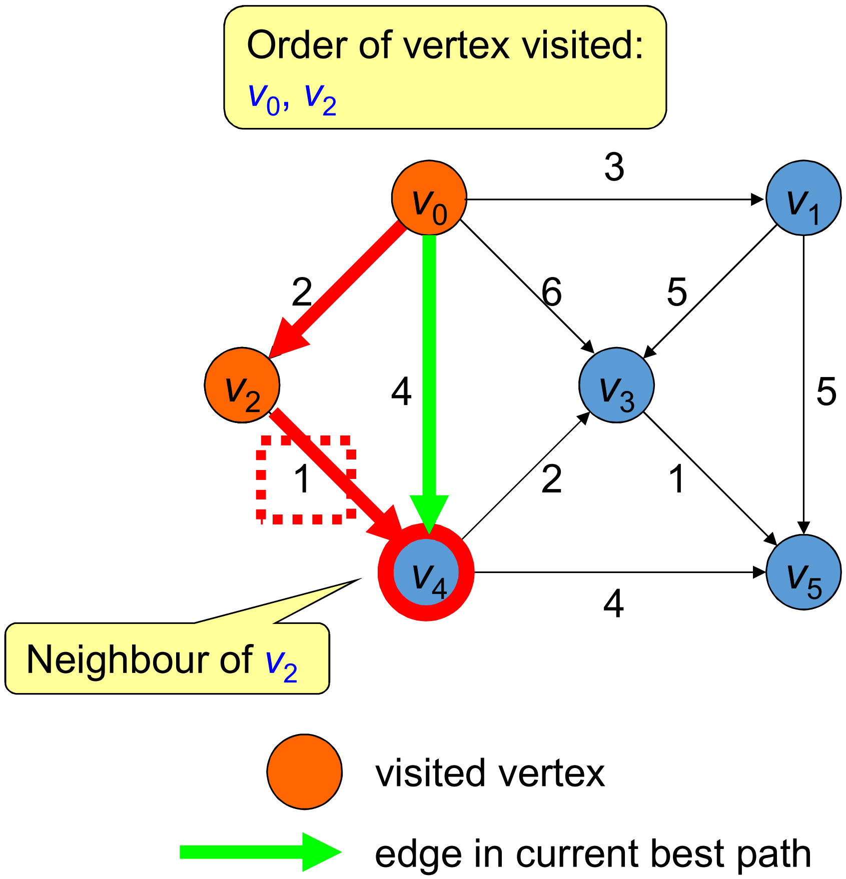
v_2

For v_4

Old path : cost/distance = 4

New path : path from v_0 to v_2 and then from v_2 to v_4 =2 + 1

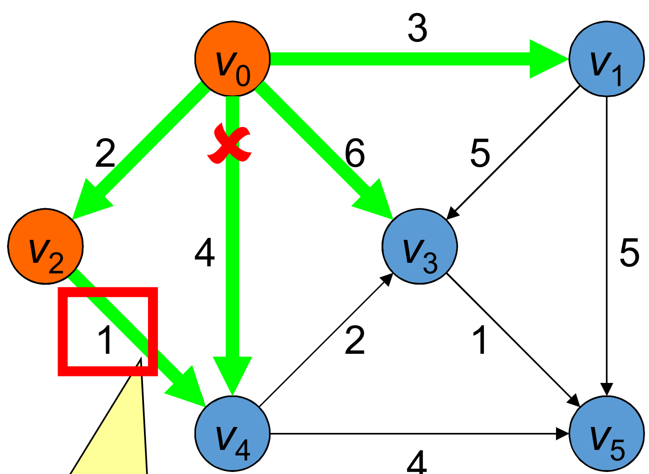
Compare the cost of the old path with that of the new path



vertex	distance	previous
v_0	0	0
v_1	3	v_0
v_2	2	v_0
v_3	6	v_0
v_4	4	v_0
v_5	∞	0

Dijkstra's Algorithm

Order of vertex visited:
v_0, v_2



Better path found

Current best cost from v_0

vertex	distance	previous
v_0	0	0
v_1	3	v_0
v_2	$\boxed{2}$	v_0
v_3	6	v_0
v_4	~~4~~ 3	~~v_0~~ v_2
v_5	∞	0

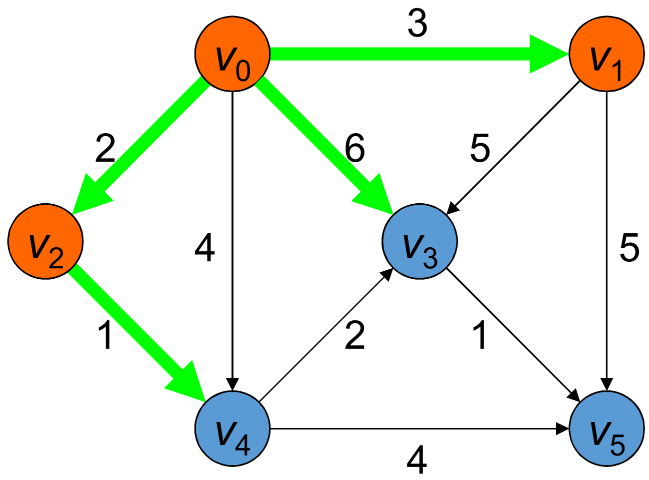
Update distance = dist[v_2] + edge[v_2, v_4] = 2 + 1 = 3 < 4

visited vertex

edge in current best path

Dijkstra's Algorithm

Order of vertex visited:
v_0, v_2, v_1



Current best cost from v_0

vertex	distance	previous
v_0	0	0
v_1	3	v_0
v_2	2	v_0
v_3	6	v_0
v_4	3	v_2
v_5	∞	0

Next vertex with min cost

visited vertex

edge in current best path

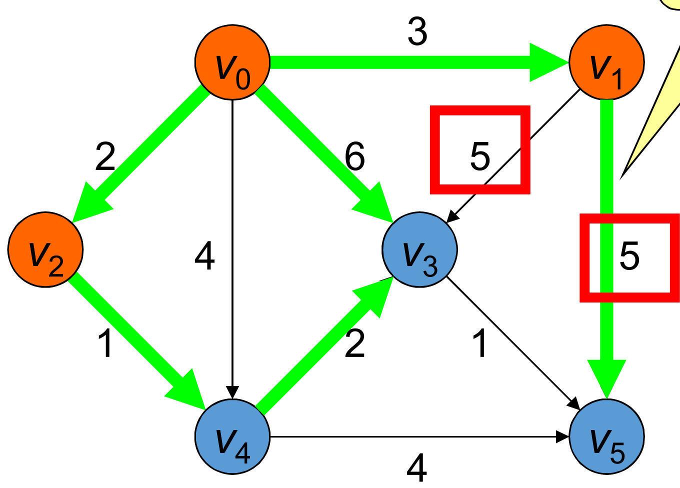
Pick the unvisited vertex with minimum current distance. → v_1
Check the adjacent vertices [v_3, v_5] if there are better paths.

v_1

Dijkstra's Algorithm

Order of vertex visited: v_0, v_2, v_1

Better path found



visited vertex

edge in current best path

Current best cost from v_0

vertex	distance	prev
v_0	0	0
v_1	$\boxed{3}$	v_0
v_2	2	v_0
v_3	6	v_0
v_4	3	v_2
v_5	$\cancel{\infty}$ 8	$\cancel{0}$ v_1

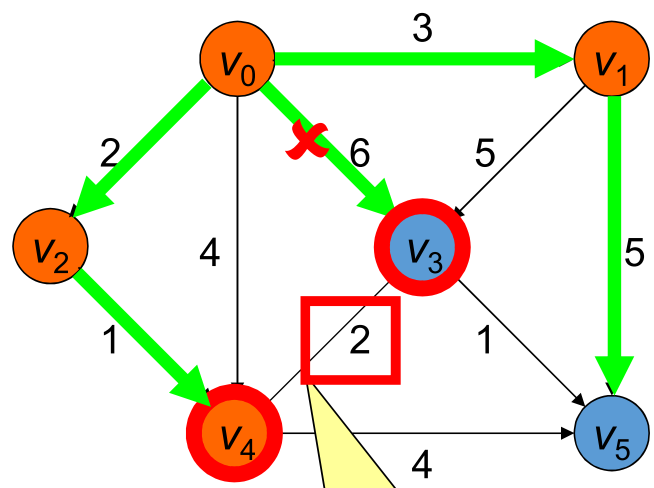
No update
dist[v_1] + edge[v_1, v_3]
= 3 + 5 = 8 > 6

Update distance =
dist[v_1] + edge[v_1, v_5]
= 3 + 5 = 8 < ∞

Pick the unvisited vertex with minimum current distance. → v_1
Check the adjacent vertices [v_3, v_5] if there are better paths.

Dijkstra's Algorithm

Order of vertex visited:
v_0, v_2, v_1, v_4



Better path found

visited vertex

edge in current best path

Current best cost from v_0

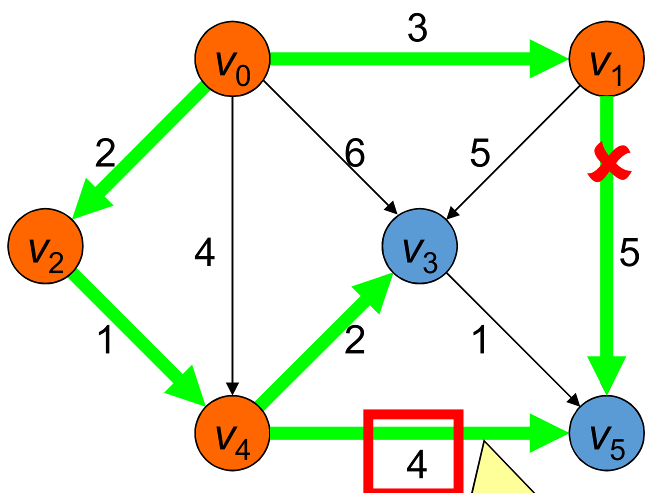
Update distance

vertex	distance	previous
v_0	0	0
v_1	3	v_0
v_2	2	v_0
v_3	~~6~~ 5	~~v_0~~ v_4
v_4	$\boxed{3}$	v_2
v_5	8	v_1

Pick the unvisited vertex with minimum current distance. → v_4
Check the adjacent vertices [v_3, v_5] if there are better paths.

Dijkstra's Algorithm

Order of vertex visited:
v_0, v_2, v_1, v_4



Better path found

visited vertex

edge in current best path

Current best cost from v_0

vertex	distance	previous
v_0	0	0
v_1	3	v_0
v_2	2	v_0
v_3	~~6~~ 5	~~v_4~~ v_4
v_4	$\boxed{3}$	v_2
v_5	~~8~~ 7	~~v_1~~ v_4

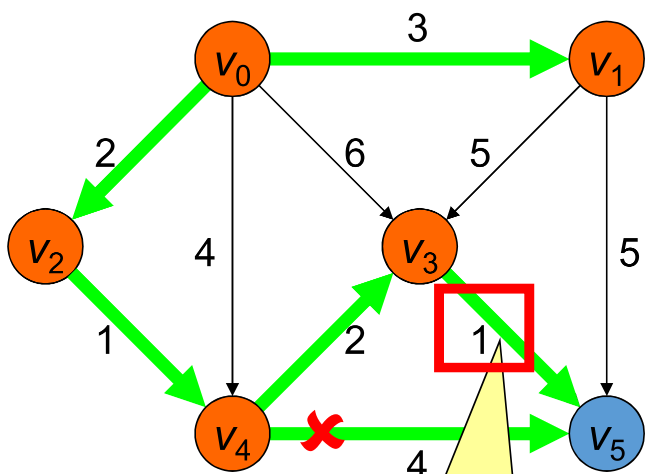
Update distance

Pick the unvisited vertex with minimum current distance. → v_4
Check the adjacent vertices [v_3, v_5] if there are better paths.

v_3

Dijkstra's Algorithm

Order of vertex visited:
v_0, v_2, v_1, v_4, v_3



Better path found

Current best cost from v_0

vertex	distance	previous
v_0	0	0
v_1	3	v_0
v_2	2	v_0
v_3	$\boxed{5}$	v_4
v_4	3	v_2
v_5	~~7~~ 6	~~v_4~~ v_3

Update distance

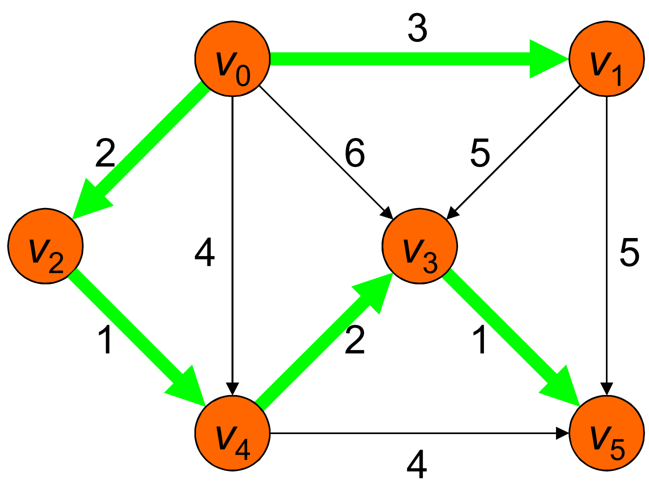
visited vertex

edge in current best path

Pick the unvisited vertex with minimum current distance.
Check that vertex if there are better paths.

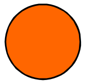
Dijkstra’s Algorithm

Order of vertex visited:
v_0, v_2, v_1, v_4, v_3, v_5



Done!

vertex	distance	previous
v_0	0	0
v_1	3	v_0
v_2	2	v_0
v_3	5	v_4
v_4	3	v_2
v_5	6	v_3

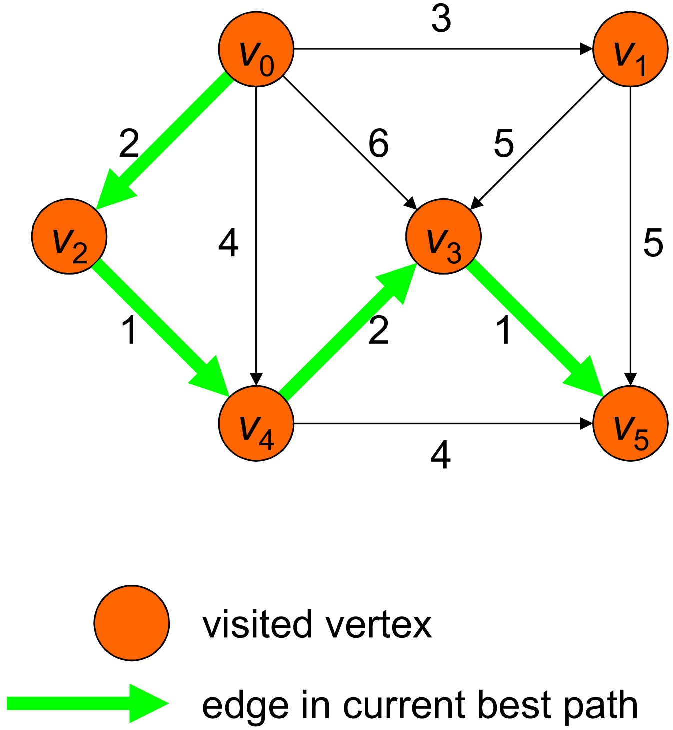

visited vertex


edge in current best path

During the algorithm, a visited vertex v_j means the shortest path from v_0 (start) to v_j has been found already.

Dijkstra's Algorithm

Order of vertex visited:
v_0, v_2, v_1, v_4, v_3, v_5



Only the previous vertex is needed

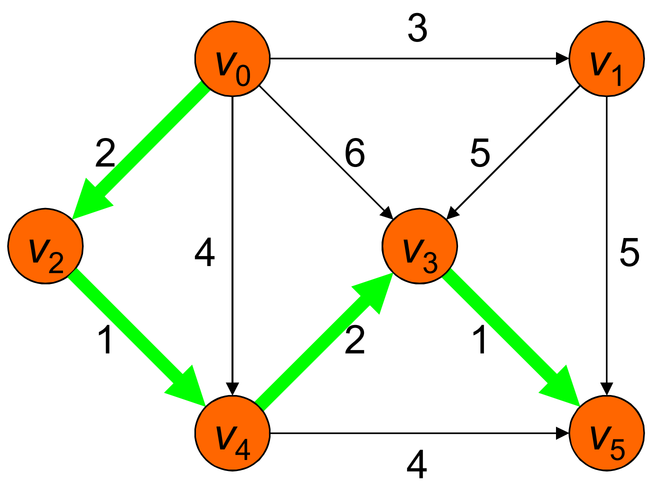
vertex	distance	previous
v_0	0	0
v_1	3	v_0
v_2	2	v_0
v_3	5	v_4
v_4	3	v_2
v_5	6	v_3

path[v_5] = … v_3 v_5

Dijkstra’s Algorithm

Only the previous vertex is needed

Order of vertex visited:
v_0, v_2, v_1, v_4, v_3, v_5



vertex	distance	previous
v_0	0	0
v_1	3	v_0
v_2	2	v_0
v_3	5	v_4
v_4	3	v_2
v_5	6	v_3

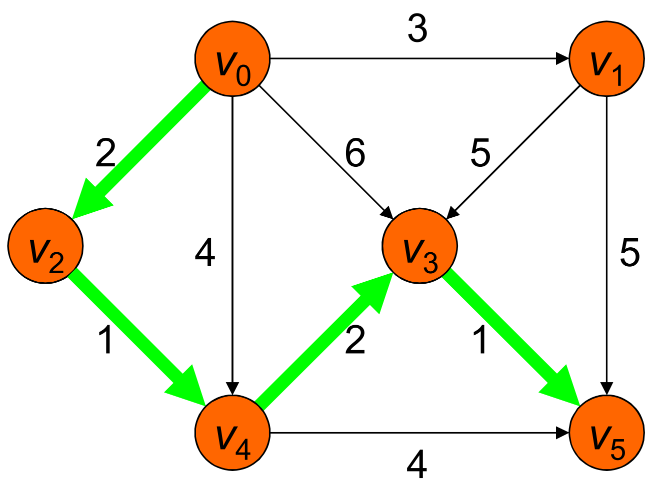
path[v_5] = … v_4 v_3 v_5

visited vertex

edge in current best path

Dijkstra's Algorithm

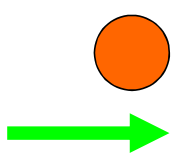
Order of vertex visited:
v_0, v_2, v_1, v_4, v_3, v_5



Only the previous vertex is needed

vertex	distance	previous
v_0	0	0
v_1	3	v_0
v_2	2	v_0
v_3	5	v_4
v_4	3	v_2
v_5	6	v_3

path[v_5] = … v_2 v_4 v_3 v_5



visited vertex

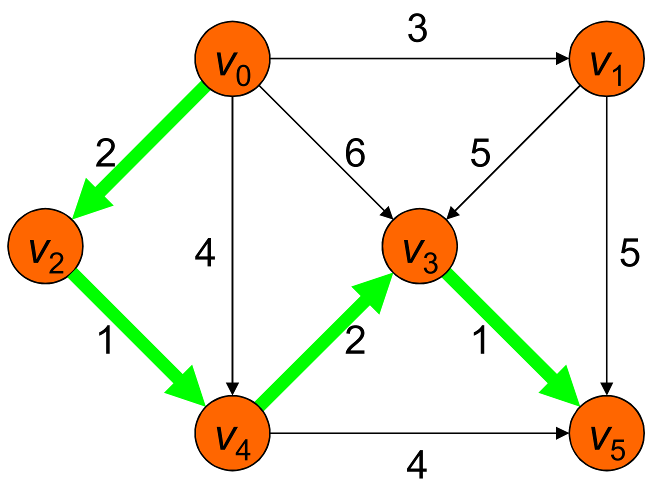
edge in current best path

Dijkstra's Algorithm

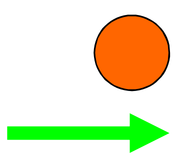
path[v_5] = v_0 v_2 v_4 v_3 v_5

Only the previous vertex is needed

Order of vertex visited:
v_0, v_2, v_1, v_4, v_3, v_5



vertex	distance	previous
v_0	0	0
v_1	3	v_0
v_2	2	v_0
v_3	5	v_4
v_4	3	v_2
v_5	6	v_3



visited vertex

edge in current best path

Dijkstra's Algorithm

- Initialize the cost of each vertex = infinity, except the source vertex = 0
- While not all vertices are visited
  - Pick the unvisited vertex *v* with the lowest cost and mark it as visited.
  - For any other unvisited vertex *u* adjacent to *v*
  - if cost[v] + edge[u,v] < cost[u]

    cost[u] = cost[v] + edge[u,v]

    Update u's previous vertex as v
- Reconstruct path from target back to source using the previous vertices

Miscellaneous

- Dijkstra's algorithm works for graphs with *non-negative* edge weights only.
  - Other shortest path algorithms, e.g., Bellman-Ford's algorithm, can be used otherwise.

- The shortest path problem does not make sense if a graph contains *negative cycles*.