CS 2124: DATA STRUCTURES
Spring 2024

Lecture 12

Topics: Breadth First Search (BFS), Depth First Search (DFS), and Dijkstra's Algorithm

TOPICS

1. Graph Traversal
   I. DFS (Depth-first search)
      - Implementation
   II. BFS (Breadth-first search)
      - Implementation
2. Graph Searching Implementation in Game Programming Cases Using BFS and DFS Algorithms
3. Spanning Trees
   I. Spanning Tree example/case
4. MST (Minimum Spanning Tree)
   I. Kruskal Algorithm
   II. Prim’s Algorithm
5. MST – Applications
6. Single-Source Shortest Path Problem (SSSP)
7. Dijkstra's algorithm
   I. Applications of Dijkstra's Algorithm
8. A Gentle Introduction to Graph Neural Networks

Graph Traversal Algorithm

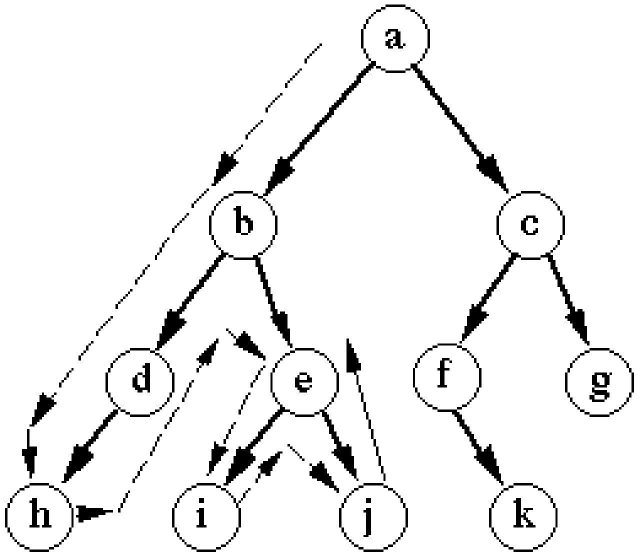
- Graph traversal is a search technique for finding a vertex in a graph.
- In the search process, graph traversal is also used to determine the order in which it visits the vertices.
- Without producing loops, a graph traversal finds the edges to be employed in the search process.

- There are two methods to traverse a graph data structure:
    1. Depth-First Search or DFS algorithm
    2. Breadth-First Search or BFS algorithm

Graph Traversal

Depth-first search (DFS)

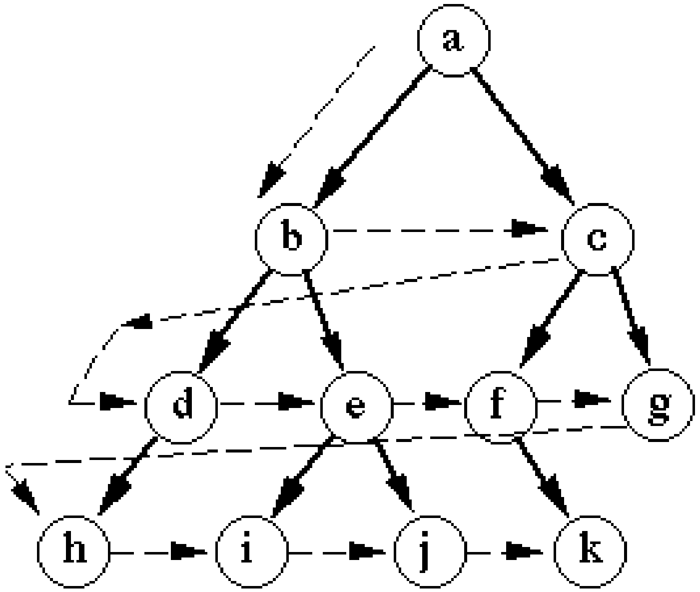
- DFS goes through a graph as far as possible in one direction before backtracking to other nodes.
- DFS is similar to the pre-order tree traversal, but you need to make sure you don't get stuck in a loop.
- To do this, you'll need to keep track of which Nodes have been visited.



Depth-first search

Breadth-first search (BFS)

- BFS is a graph traversal algorithm that explores nodes in the order of their distance from the roots, where distance is defined as the minimum path length from a root to the node.



Breadth-first search

Depth-first search (DFS)

- Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph.
- A standard DFS implementation puts each vertex of the graph into one of two categories:
  1. Visited
  2. Not Visited
- The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

- The DFS algorithm works as follows (Stack based):
  1. Start by putting any one of the graph's vertices on top of a stack.
  2. Take the top item of the stack and add it to the visited list.
  3. Create a list of that vertex's adjacent nodes.
     I. Add the ones which aren't in the visited list to the top of the stack.
  4. Keep repeating steps 2 and 3 until the stack is empty.

Graph Traversal
Depth-first search (DFS) for Graphs

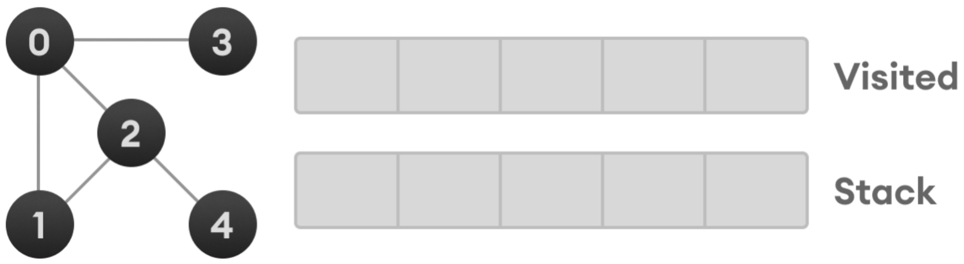
- **Concept:** DFS algorithm is a recursive algorithm that uses the backtracking principle. It entails conducting exhaustive searches of all nodes by moving forward if possible and backtracking, if necessary.

- **Stack based implementation:** To visit the next node, pop the top node from the stack and push all of its nearby nodes into a stack.

- **Applications:** Topological sorting, scheduling problems, graph cycle detection, and solving puzzles with just one solution, such as a maze or a sudoku puzzle, all employ depth-first search algorithms. Other applications include network analysis, such as determining if a graph is bipartite (vertices of that graph can be divided into two independent sets).

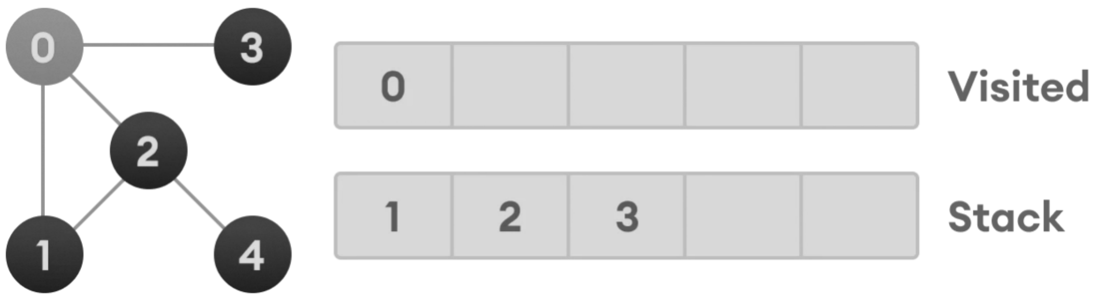
Depth-first search (DFS)

1



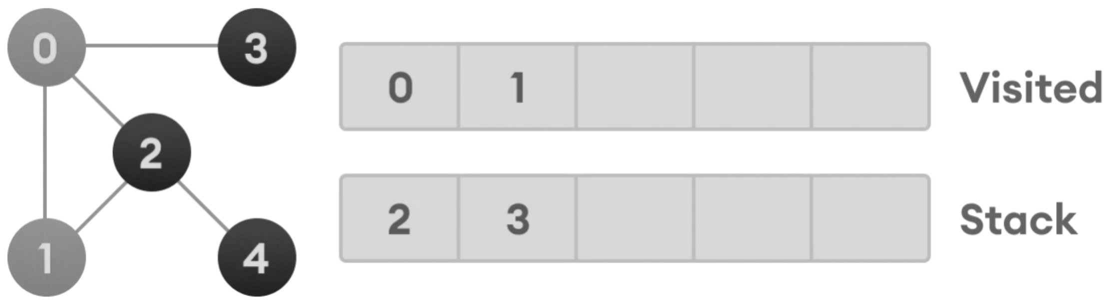
Undirected graph with 5 vertices

2



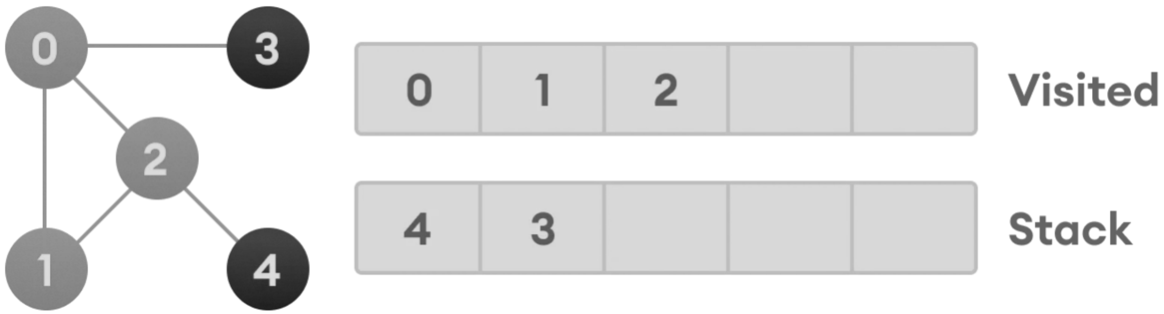
Start from vertex 0, the DFS algorithm starts by putting it in the visited list and putting all its adjacent vertices in the stack.

3



Next, visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead

4



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

Continue next slide →

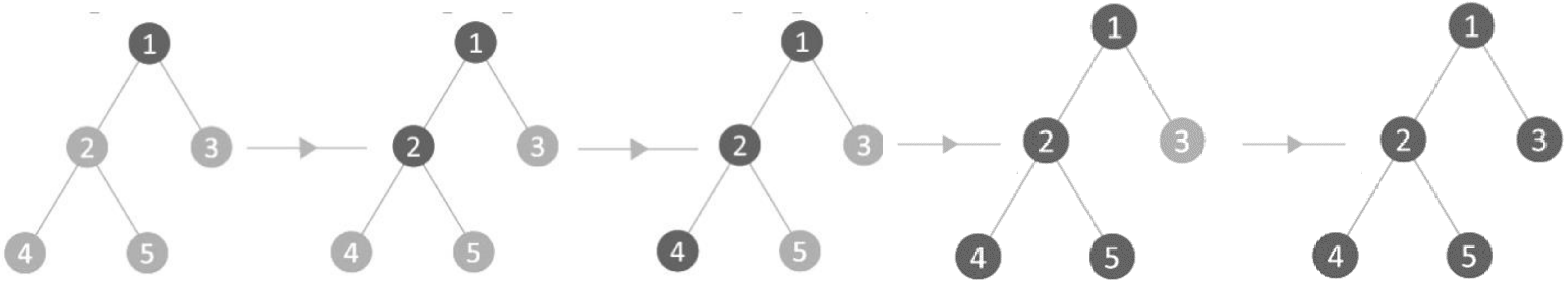
Depth-first search (DFS)

5

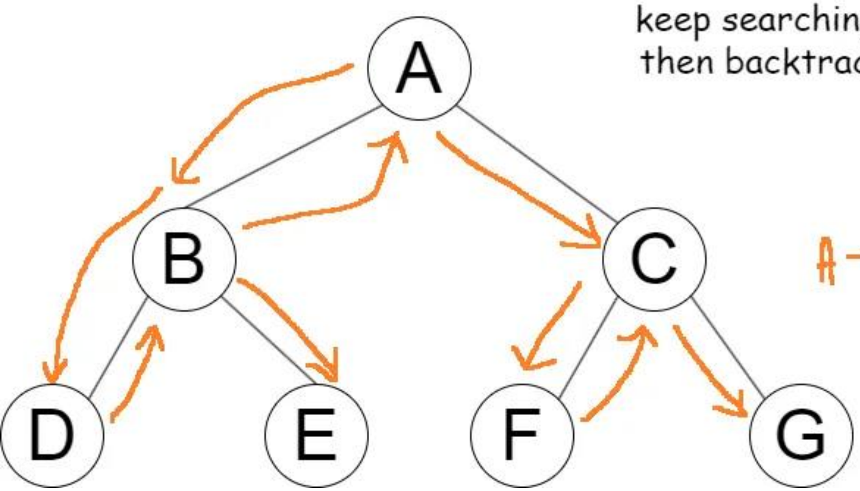
After visiting the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph

6

Traversal Complete

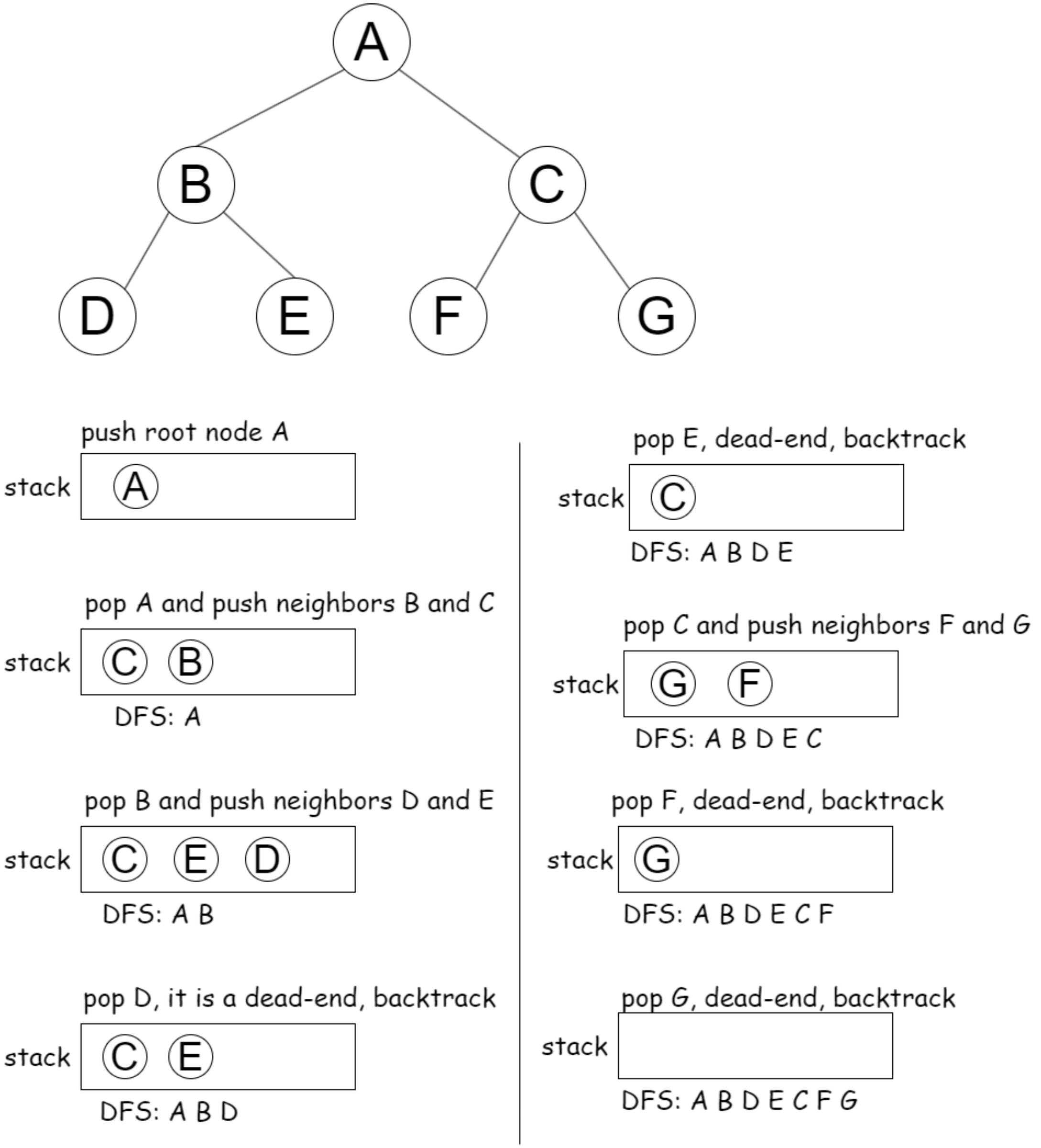


Depth-first search (DFS)



keep searching a route until hit dead-end,
then backtrack and search another route

A→B→D→E→C→F→G



Source

Depth-first search (DFS)

Implementation 1/2

```c
#include <stdio.h>
#include <stdlib.h>
struct node {
  int vertex;
  struct node* next;
};
struct node* createNode(int v);
struct Graph {
  int numVertices;
  int* visited; //int** to store a two dimensional array.
  struct node** adjLists; //node** to store an array of Linked lists
};
void DFS(struct Graph* graph, int vertex) { // DFS algo
  struct node* adjList = graph->adjLists[vertex];
  struct node* temp = adjList;
  graph->visited[vertex] = 1;
  printf("Visited %d \n", vertex);
  while (temp != NULL) {
    int connectedVertex = temp->vertex;
    if (graph->visited[connectedVertex] == 0) {
      DFS(graph, connectedVertex);
    }
    temp = temp->next;
  }
}
```

```c
struct node* createNode(int v) { // Create a node
  struct node* newNode = malloc(sizeof(struct node));
  newNode->vertex = v;
  newNode->next = NULL;
  return newNode;
}
struct Graph* createGraph(int vertices) { // Create graph
  struct Graph* graph = malloc(sizeof(struct Graph));
  graph->numVertices = vertices;
  graph->adjLists = malloc(vertices * sizeof(struct node*));
  graph->visited = malloc(vertices * sizeof(int));
  int i;
  for (i = 0; i < vertices; i++) {
    graph->adjLists[i] = NULL;
    graph->visited[i] = 0;
  }
  return graph;
}
void addEdge(struct Graph* graph, int src, int dest) { // Add edge
  struct node* newNode = createNode(dest); // Add edge from src to dest
  newNode->next = graph->adjLists[src];
  graph->adjLists[src] = newNode;
  newNode = createNode(src); // Add edge from dest to src
  newNode->next = graph->adjLists[dest];
  graph->adjLists[dest] = newNode;
}
```

Note: This code will not be part of quiz or exam. It is only for implementation and understanding

Depth-first search (DFS)

Implementation 2/2

```c
void printGraph(struct Graph* graph) { //Print the graph
  int v;
  for (v = 0; v < graph->numVertices; v++) {
    struct node* temp = graph->adjLists[v];
    printf("\n Adjacency list of vertex %d\n ", v);
    while (temp) {
      printf("%d -> ", temp->vertex);
      temp = temp->next;
    }
    printf("\n");
  }
}
int main() {
  struct Graph* graph = createGraph(4);
  addEdge(graph, 0, 1);
  addEdge(graph, 0, 2);
  addEdge(graph, 1, 2);
  addEdge(graph, 2, 3);
  printGraph(graph);
  DFS(graph, 2);
  return 0;
}
```

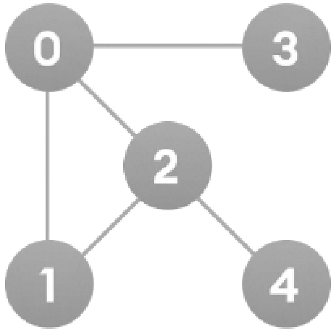
```
 Adjacency list of vertex 0
 2 -> 1 ->

 Adjacency list of vertex 1
 2 -> 0 ->

 Adjacency list of vertex 2
 3 -> 1 -> 0 ->

 Adjacency list of vertex 3
 2 ->
Visited 2
Visited 3
Visited 1
Visited 0
```

Note: This code will not be part of quiz or exam. It is only for implementation and understanding

Depth-first search (DFS)

Implementation based on the graph discussed on slides 7 & 8

```c
int main() {
   struct Graph* graph = createGraph(5);
   addEdge(graph, 0, 1);
   addEdge(graph, 0, 2);
   addEdge(graph, 0, 3);
   addEdge(graph, 1, 2);
   addEdge(graph, 2, 4);
   printGraph(graph);
   DFS(graph, 0);
   return 0;
}
```




```
 Adjacency list of vertex 0
 3 -> 2 -> 1 ->

 Adjacency list of vertex 1
 2 -> 0 ->

 Adjacency list of vertex 2
 4 -> 1 -> 0 ->

 Adjacency list of vertex 3
 0 ->

 Adjacency list of vertex 4
 2 ->
Visited 0
Visited 3
Visited 2
Visited 4
Visited 1
```

- Graph based on the visual representation on slides 7 & 8
- Using the same based code as on slides 9 & 10

Note: This code will not be part of quiz or exam. It is only for implementation and understanding

Breadth-first search (BFS)

- A standard BFS implementation puts each vertex of the graph into one of two categories:
  - Visited
  - Not Visited
- The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

- The algorithm works as follows (Queue Based):
  1. Start by putting any one of the graph's vertices at the back of a queue.
  2. Take the front item of the queue and add it to the visited list.
  3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
  4. Keep repeating steps 2 and 3 until the queue is empty.
  5. The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

Graph Traversal
Breadth-first search (BFS) for Graphs

- **Concept:** BFS algorithm is used to search a tree or graph data structure for a node that meets a set of criteria. You start at a source node and layer by layer through the graph, analyzing the nodes directly related to the source node. Then, in BFS traversal, you must move on to the next-level neighbor nodes.

- **Working:** It begins at the root of the tree or graph and investigates all nodes at the current depth level before moving on to nodes at the next depth level.

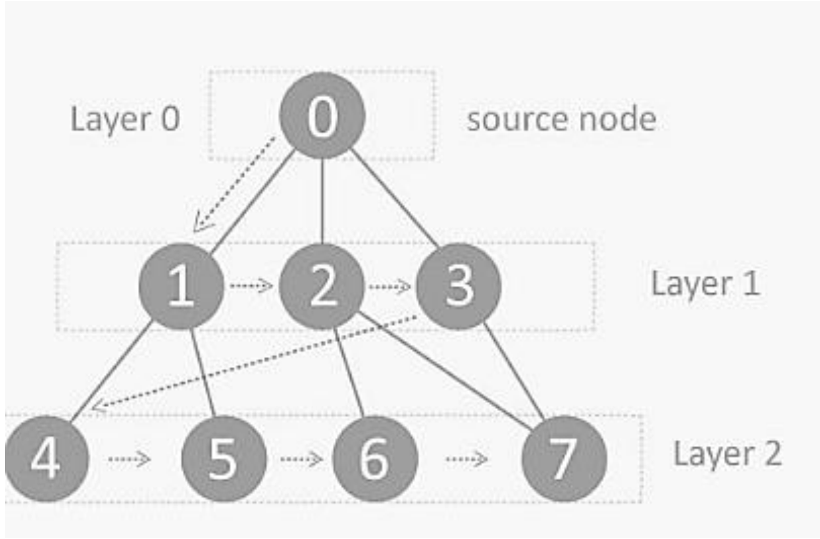
- **Example:** You can solve many problems in graph theory via the BFS. For example, finding the shortest path between two vertices a and b is determined by the number of edges. In a flow network, the Ford–Fulkerson method is used to calculate the maximum flow and when a binary tree is serialized/deserialized* instead of serialized in sorted order, the tree can be reconstructed quickly.

- *Serializing a binary tree is done by storing the preorder or postorder traversal sequence of the tree by maintaining a marker to null nodes.*
- *Deserialization of a binary tree from the given sequence is done by recreating the tree by following the corresponding traversal manner.*

Graph Traversal
Breadth-first search (BFS) for Graphs

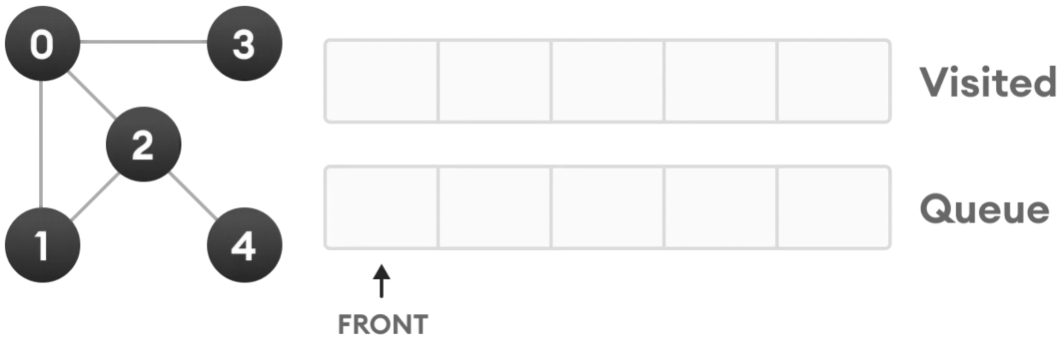
- **Rules** to Remember in the BFS Algorithm
  1. You can take any node as your source node or root node.
  2. You should explore all the nodes.
  3. And don't forget to explore on repeated nodes.
  4. You must transverse the graph in a breadthwise direction, not depth wise.

- **Architecture** of the BFS Algorithm
  1. We are allowed to use any node as our source node as per the law
  2. Then we explore breadthwise and find the nodes which are adjacently connected to our source node.



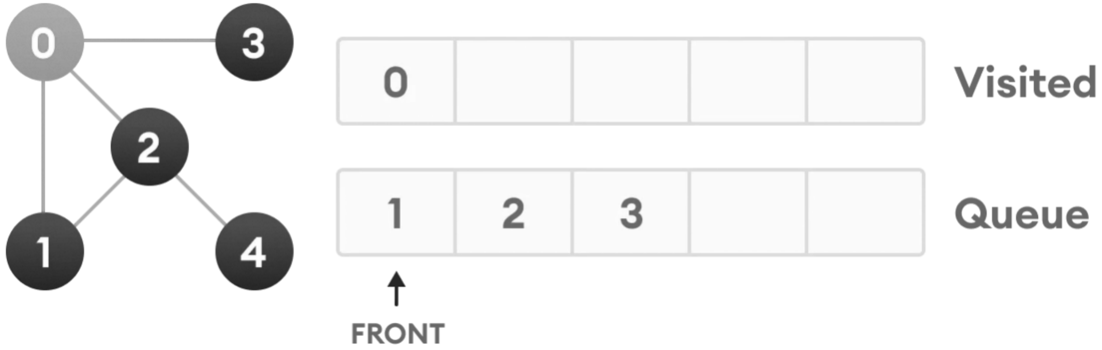
Breadth-first search (BFS)

1



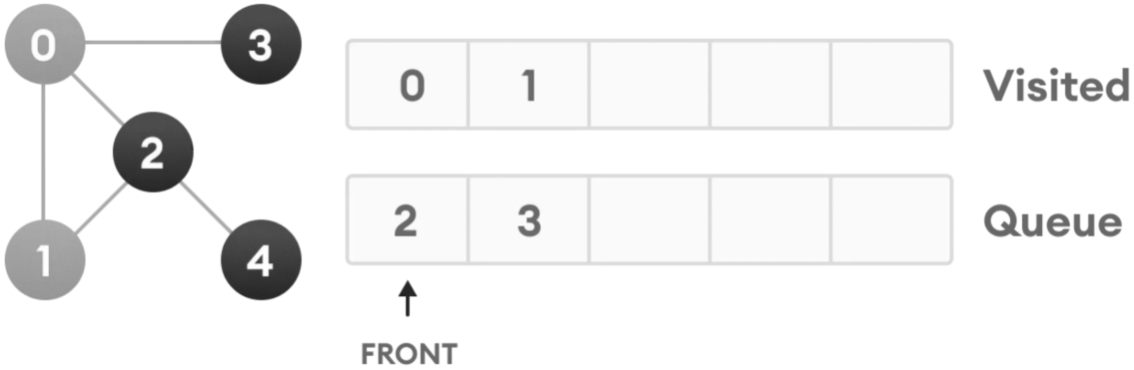
Start from vertex 0, the BFS algorithm starts by putting it in the visited list and putting all its adjacent vertices in the stack

2



Visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

3



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue

4

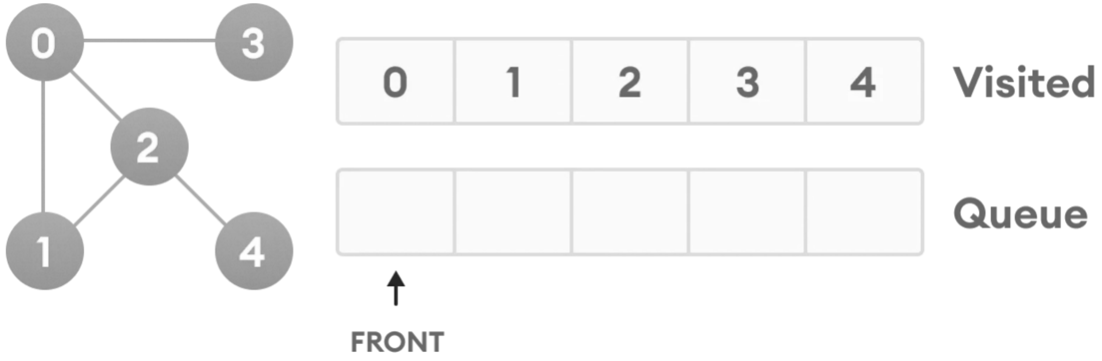
Continue next slide →

Breadth-first search (BFS)

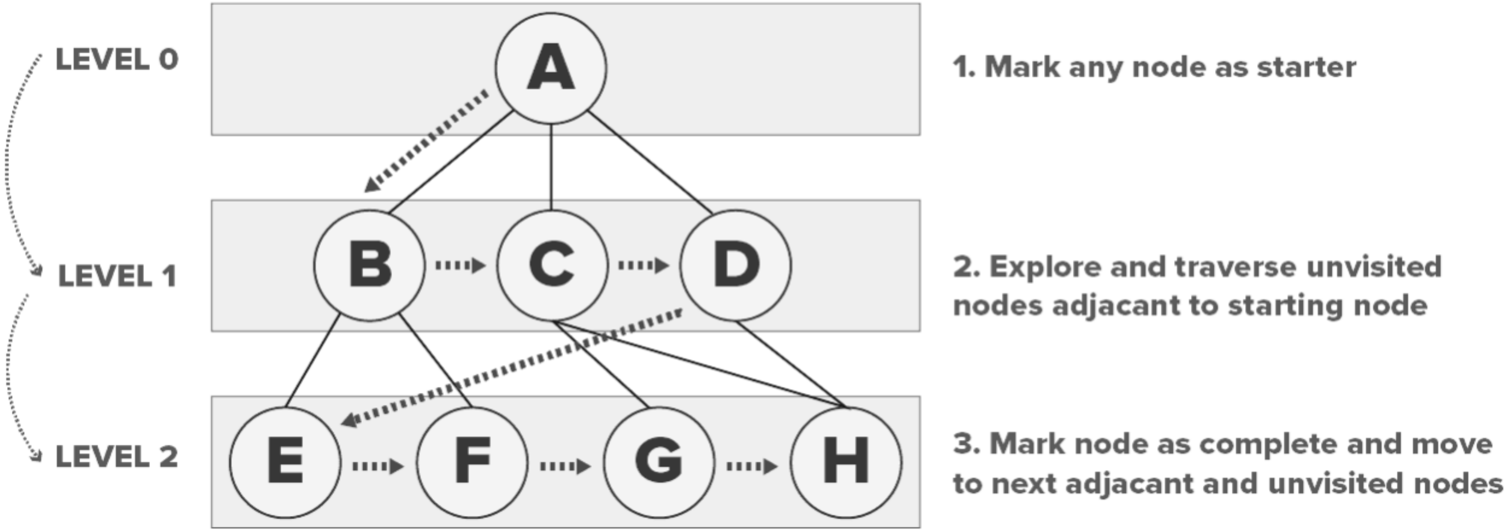
5

Only 4 remains in the queue since the only adjacent node of 3 i.e. 0 is already visited.

6



Traversal Complete



Breadth-first search (BFS)

Implementation 1/3

```c
#include <stdio.h>
#include <stdlib.h>
#define SIZE 10
struct queue {
  int items[SIZE];
  int front;
  int rear;
};
struct queue* createQueue();
void enqueue(struct queue* q, int);
int dequeue(struct queue* q);
void display(struct queue* q);
int isEmpty(struct queue* q);
void printQueue(struct queue* q);
struct node {
  int vertex;
  struct node* next;
};
struct node* createNode(int);
struct Graph {
  int numVertices;
  struct node** adjLists;
  int* visited;
};
```

```c
void bfs(struct Graph* graph, int startVertex) {
  struct queue* q = createQueue(); // BFS algorithm
  graph->visited[startVertex] = 1;
  enqueue(q, startVertex);
  while (!isEmpty(q)) {
    printQueue(q);
    int currentVertex = dequeue(q);
    printf("Visited %d\n", currentVertex);
    struct node* temp = graph->adjLists[currentVertex];
    while (temp) {
      int adjVertex = temp->vertex;
      if (graph->visited[adjVertex] == 0) {
        graph->visited[adjVertex] = 1;
        enqueue(q, adjVertex);
      }
      temp = temp->next;
    }  }  }
struct node* createNode(int v) { // Creating a node
  struct node* newNode = malloc(sizeof(struct node));
  newNode->vertex = v;
  newNode->next = NULL;
  return newNode;
}
```

Note: This code will not be part of quiz or exam. It is only for implementation and understanding

Breadth-first search (BFS)

Implementation 2/3

```c
struct Graph* createGraph(int vertices) { // Creating a graph
  struct Graph* graph = malloc(sizeof(struct Graph));
  graph->numVertices = vertices;
  graph->adjLists = malloc(vertices * sizeof(struct node*));
  graph->visited = malloc(vertices * sizeof(int));
  int i;
  for (i = 0; i < vertices; i++) {
    graph->adjLists[i] = NULL;
    graph->visited[i] = 0;
  }
  return graph;
} // Add edge
void addEdge(struct Graph* graph, int src, int dest) {
  // Add edge from src to dest
  struct node* newNode = createNode(dest);
  newNode->next = graph->adjLists[src];
  graph->adjLists[src] = newNode;
  // Add edge from dest to src
  newNode = createNode(src);
  newNode->next = graph->adjLists[dest];
  graph->adjLists[dest] = newNode;
}
struct queue* createQueue() { // Create a queue
  struct queue* q = malloc(sizeof(struct queue));
  q->front = -1;
  q->rear = -1;
  return q;
}
```

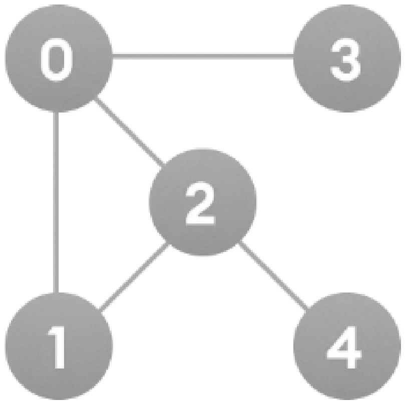
```c
int isEmpty(struct queue* q) {
  if (q->rear == -1) // Check if the queue is empty
    return 1;
  else
    return 0;
}
void enqueue(struct queue* q, int value) {
  if (q->rear == SIZE - 1) // Adding elements into queue
    printf("\nQueue is Full!!");
  else {
    if (q->front == -1)
      q->front = 0;
    q->rear++;
    q->items[q->rear] = value;
  } }
int dequeue(struct queue* q) {
  int item; // Removing elements from queue
  if (isEmpty(q)) {
    printf("Queue is empty");
    item = -1;
  } else {
    item = q->items[q->front];
    q->front++;
    if (q->front > q->rear) {
      printf("Resetting queue ");
      q->front = q->rear = -1;
    } }
  return item;
}
```

Note: This code will not be part of quiz or exam. It is only for implementation and understanding

Breadth-first search (BFS)

Implementation 3/3

```c
void printQueue(struct queue* q) {
  int i = q->front; // Print the queue
  if (isEmpty(q)) {
    printf("Queue is empty");
  } else {
    printf("\nQueue contains \n");
    for (i = q->front; i < q->rear + 1; i++) {
      printf("%d ", q->items[i]);
    }   }  }
int main() {
  struct Graph* graph = createGraph(5);
  addEdge(graph, 0, 1);
  addEdge(graph, 0, 2);
  addEdge(graph, 0, 3);
  addEdge(graph, 1, 2);
  addEdge(graph, 2, 4);
  bfs(graph, 0);
  return 0;
}
```




```
Queue contains
0 Resetting queue Visited 0

Queue contains
3 2 1 Visited 3

Queue contains
2 1 Visited 2

Queue contains
1 4 Visited 1

Queue contains
4 Resetting queue Visited 4
```

Note: This code will not be part of quiz or exam. It is only for implementation and understanding

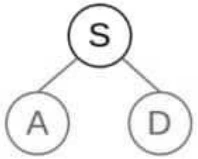
Search steps

Memory updates

Initialization

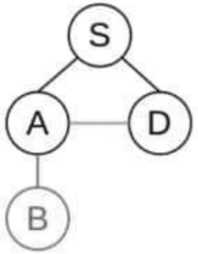
parent(S) = NULL

After expanding S



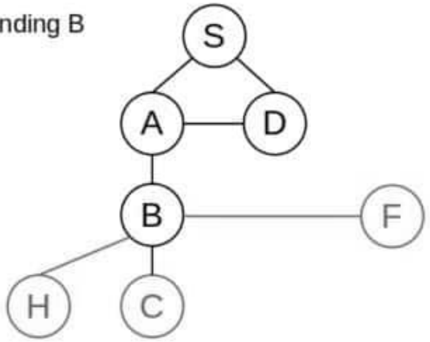
parent(A) = S
parent(D) = S

After expanding A



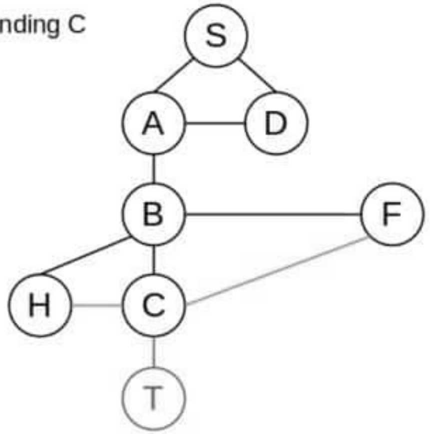
parent(B) = A
We don't update the parent of D because we already know it's S.

After expanding B



parent(F) = B
parent(C) = B
parent(H) = B

After expanding C



parent(T) = C
The parents of F and H have already been found.

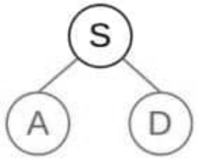
Search steps

Memory updates

Initialization

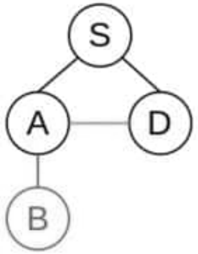
S.parent = NULL

After expanding S



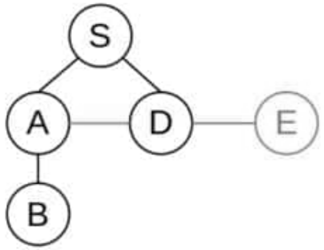
A.parent = S
D.parent = S

After expanding A



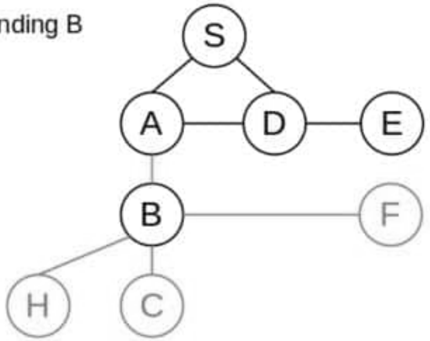
B.parent = A
We don't update the parent of D because we already know it's S.

After expanding D



E.parent = D
We don't update the parent of A.

After expanding B

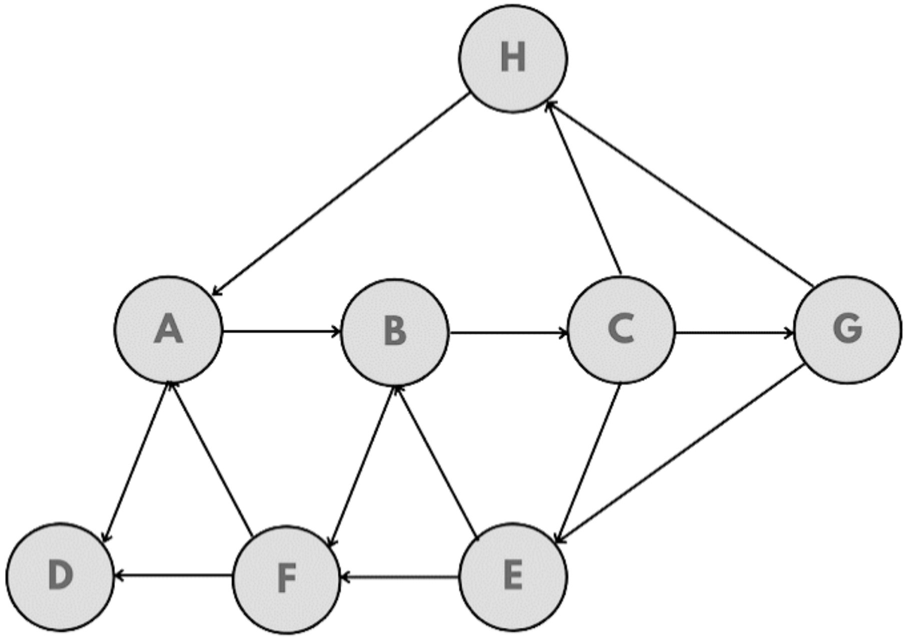


H.parent = B
C.parent = B
F.parent = B
Again, we don't update the parent of A.

We keep expanding the nodes in the BFS order until we reach the target node T.

Source

Detect Cycle using DFS (Directed Graph)



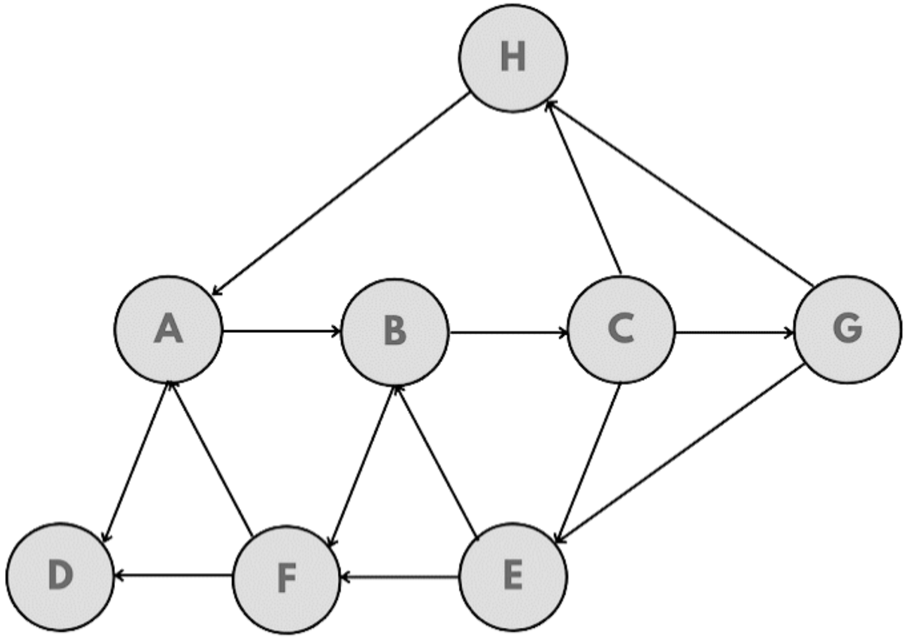
Node	Adj	Node	Adj
A	B, D	E	B, F
B	C, F	F	A
C	E, G, H	G	E, H
D	F	H	A

- DFS can be implemented using recursion or a stack data structure.
- The recursive implementation is simpler, but may not be as efficient for very large graphs.

1. Initialize all nodes as unvisited (i.e., white).
2. Pick an unvisited node and mark it as currently being explored (i.e., gray).
3. For each adjacent node of the current node:
   a) If the adjacent node is white, mark it as currently being explored (i.e., gray) and recursively visit it.
   b) If the adjacent node is gray, then a cycle has been detected.
   c) If the adjacent node is black, then it has already been fully explored, so move on to the next adjacent node.
4. Once all adjacent nodes have been visited, mark the current node as fully explored (i.e., black).
5. Repeat steps 2-4 for all unvisited nodes in the graph.

Source

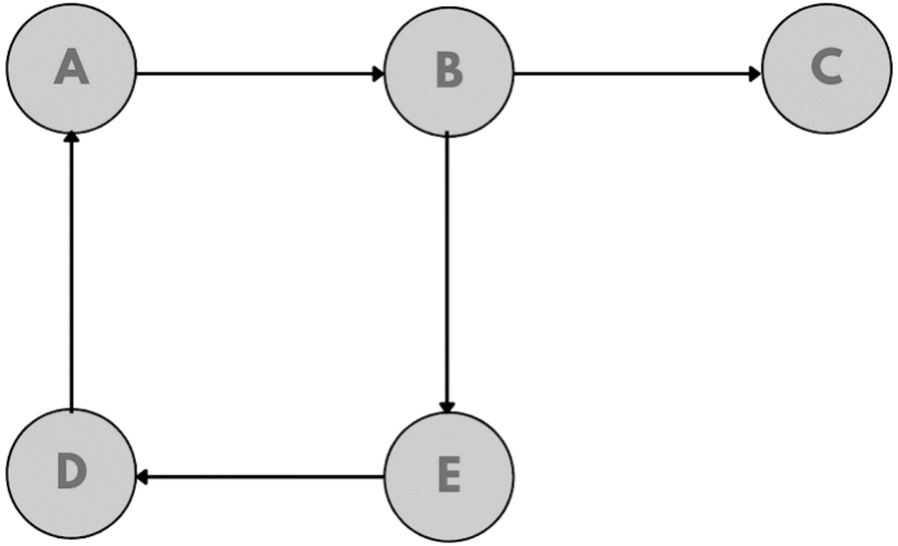
Detect Cycle using DFS (Directed Graph)



Node	Adj	Node	Adj
A	B, D	E	B, F
B	C, F	F	A
C	E, G, H	G	E, H
D	F	H	A

Source

Detect Cycle using BFS (Directed Graph)



- In this approach, we perform a BFS traversal of the graph, and if at any point we encounter a node that has already been visited and is present in the BFS queue, we can conclude that there exists a cycle in the graph.

Source

Graph Searching Implementation in Game Programming Cases Using BFS and DFS Algorithms

- **Abstract**—Graphs are heavily used in video games; hence, it is not surprising that graph searching become an essential topic in game programming. This paper will show the implementation of the most basic graph searching algorithms, the Depth-First Search (DFS) and Breadth-First Search (BFS), in some game programming cases: minesweeper, turn-based tactics, and maze games.

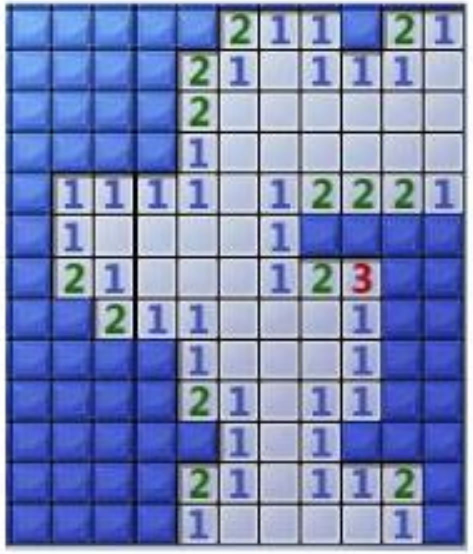
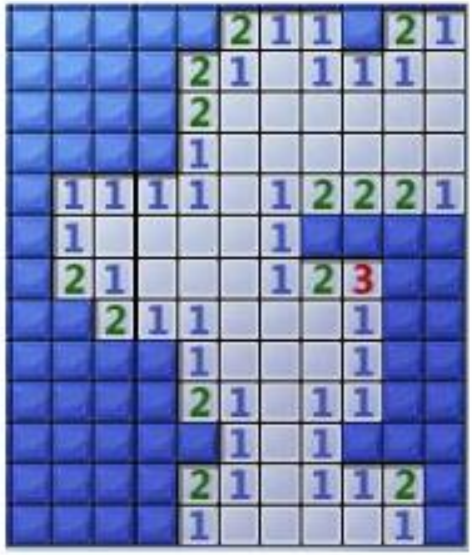


Figure 3.1 Opening an Empty Tile in Minesweeper

Figure 3.2 Available Tiles Shown as Blue Tiles.

Paper Source

Graph Searching Implementation in Game Programming Cases Using BFS and DFS Algorithms



- So how does the opening algorithm works?

- The main objective of the algorithm is to visit all empty tiles and open it.
- If it encounters a numbered tile, it opens the tile but not looks further.
- Since the main objective is to visit all tiles (or, in graph theory term, nodes), both DFS and BFS can be implemented in this problem.

DFS implementation is defined as follows:

- Create empty stack
- Mark all tiles as unvisited
- Push starting tile <i,j> to stack
- Mark <i,j> as visited
- While stack not empty
  - Pop top element to <k,l>
  - Open <k,l>
  - If tile[k,l] is empty tile then
    Check for every valid index neighboring <k,l>. If it is unvisited, push it to stack and mark it as visited.

Mark (procedure) : marks Tiles[i,j] as visited / unvisited
Open (procedure) : opens Tiles[i,j]

BFS implementation is almost exactly same as DFS one, but one needs to use queue instead of stack

[Paper Source]

Graph Searching Implementation in Game Programming Cases Using BFS and DFS Algorithms

In turn based tactics / strategy games, characters can move for a certain distance of tiles.
If player selects a character, the game shows which tiles that are available to be set on.
Tiles that are outside of character’s maximum distance, or have obstacle or other character on will not be shown as available.

So how does the coloring algorithm works?

Because of the range limitation, BFS is more suitable to be implemented than DFS as BFS visits all nodes in the same depth before visiting any nodes in the next depth.
DFS, on the other hand, may produce incorrect results because of the range limitation.

Paper Source

Graph Searching Implementation in Game Programming Cases Using BFS and DFS Algorithms

- **Abstract**—Graphs are heavily used in video games; hence, it is not surprising that graph searching become an essential topic in game programming. This paper will show the implementation of the most basic graph searching algorithms, the Depth-First Search (DFS) and Breadth-First Search (BFS), in some game programming cases: minesweeper, turn-based tactics, and maze games.

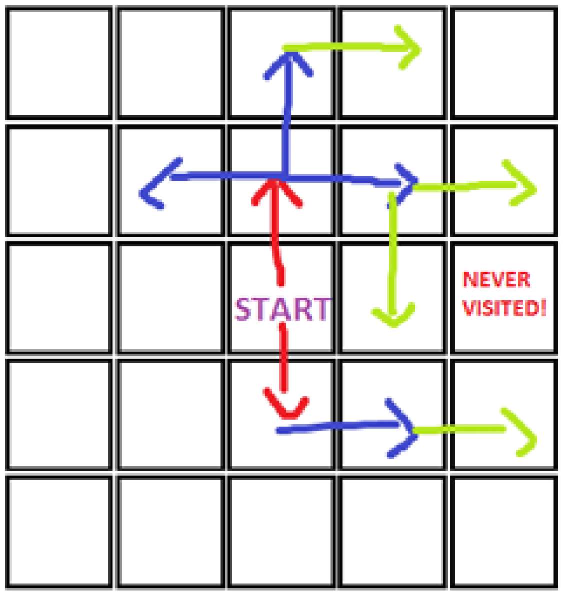


Figure 3.3 Incorrect Results in Limiting DFS Range

Figure 3.4 A Simple Area Damage Representation

Paper Source

Whoops!

404 - Page not found

One of our Development Team must be punished for this unacceptable failure!

PICK WHO TO FIRE!

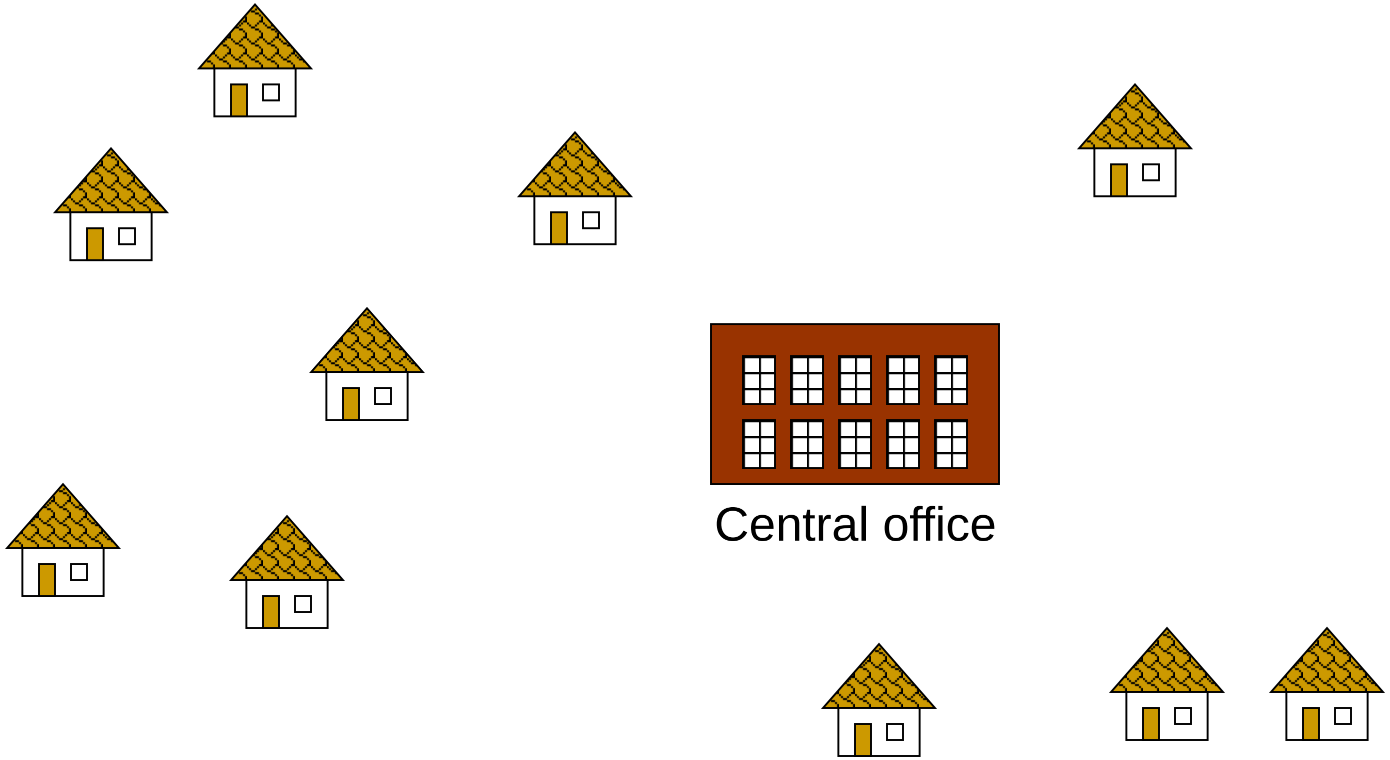
Luke | James | Martin | Chris

In a forgiving mood? Let them all keep their jobs.

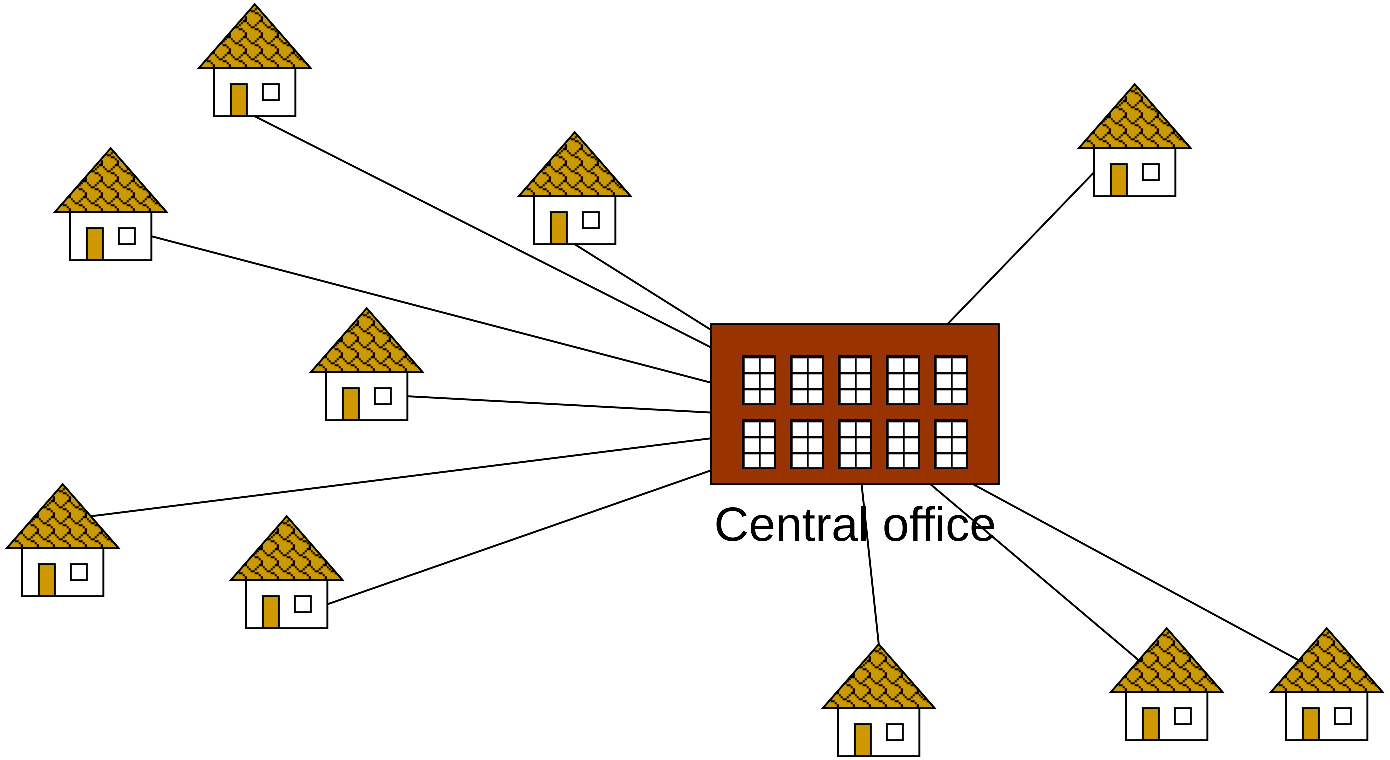
Return to the **homepage**.

Don’t have a Business Continuity Plan, consider making such page in case of an issue

Problem: Laying Telephone Wire

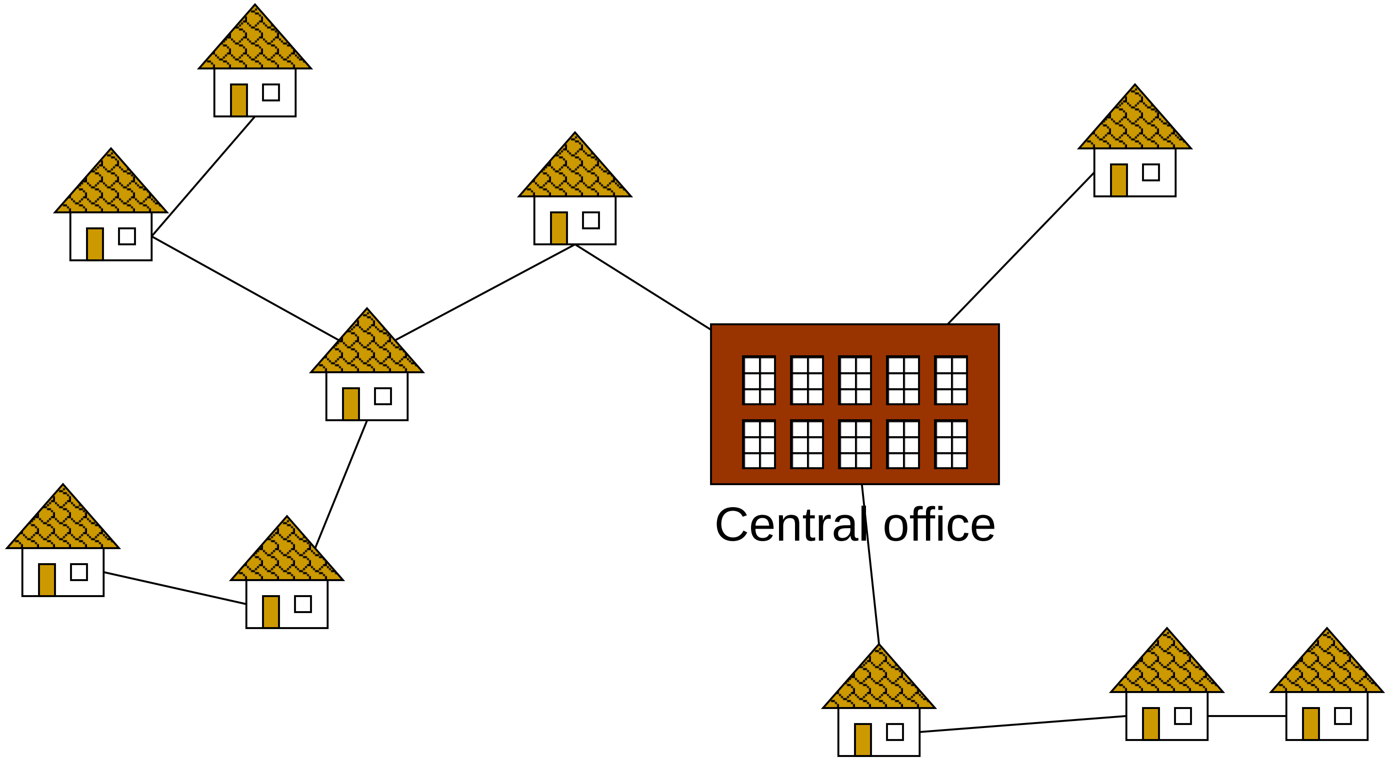


Wiring: Naïve Approach



Expensive!

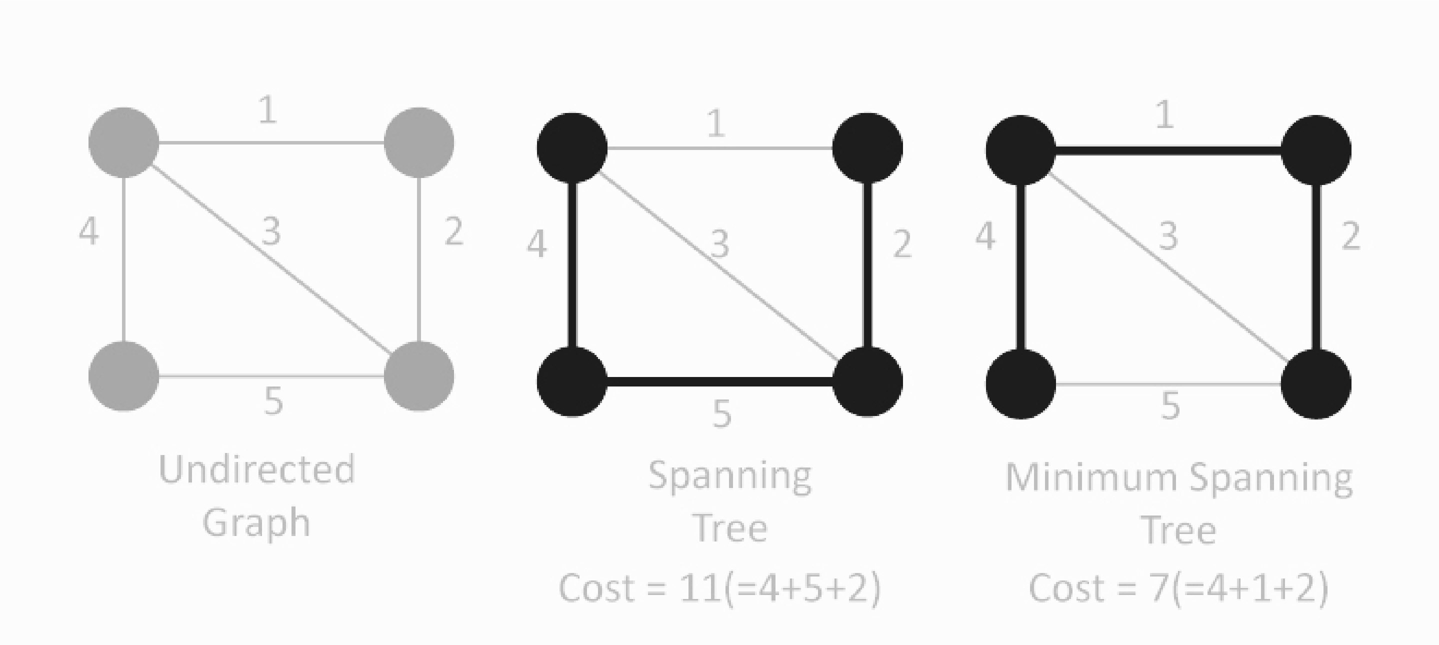
Wiring: Better Approach

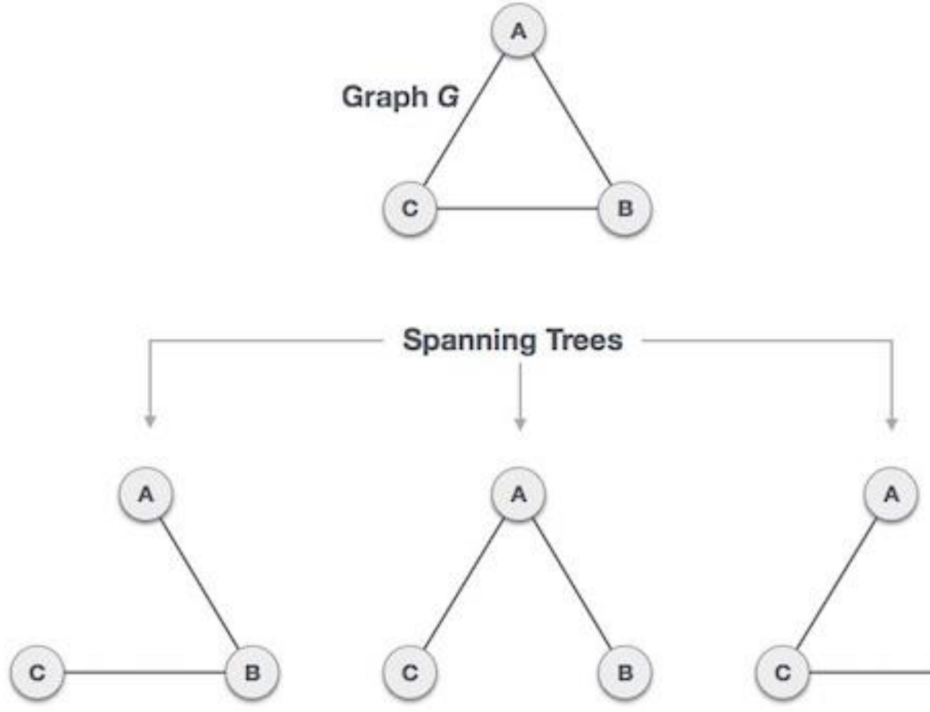


Minimize the total length of wire connecting the customers

Spanning Trees

- A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges.
- If a vertex is missed, then it is not a spanning tree.
- The edges may or may not have weights assigned to them.



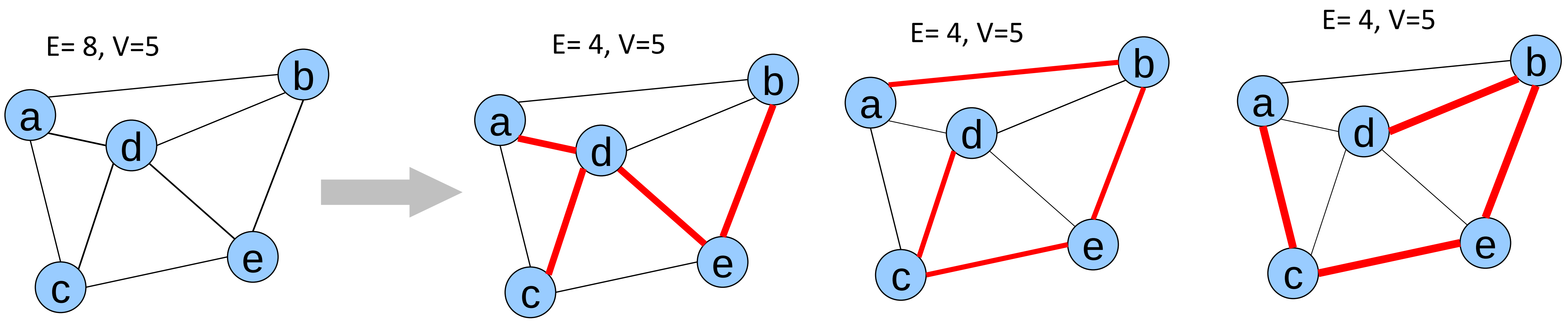


Spanning Tree
General Properties

- One graph can have more than one spanning tree.
- Following are a few properties of the spanning tree connected to graph G:
  1. A connected graph G can have more than one spanning tree.
  2. All possible spanning trees of graph G, have the same number of edges and vertices.
  3. The spanning tree does not have any cycle (loops).
  4. Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.
  5. Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.

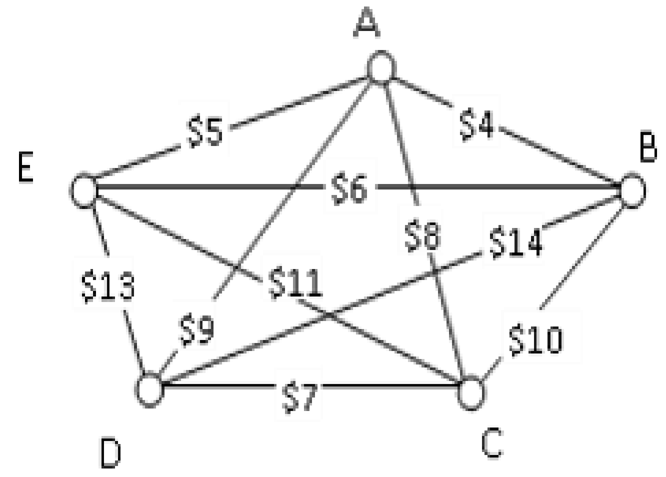
Spanning Trees

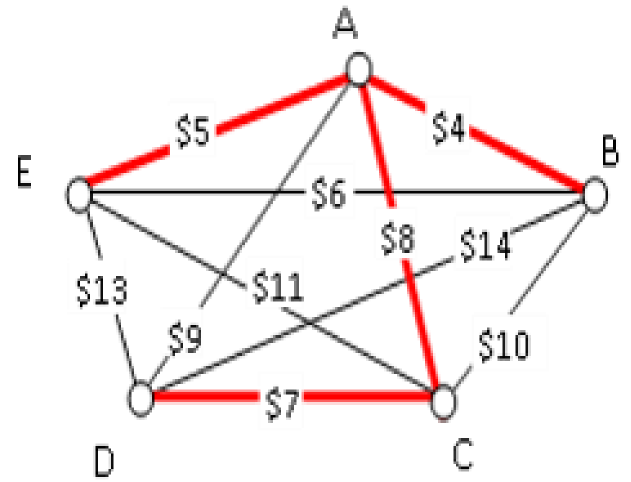
- Given (connected) graph G(V,E), A spanning tree T(V',E'):
  - Is a subgraph of G; that is, $V' \subseteq V$, $E' \subseteq E$.
  - Spans the graph (V' = V)
  - Forms a tree (no cycle);
  - So, E' has |V| -1 edges



Spanning Trees (Example Case)

- A company requires reliable internet and phone connectivity between their five offices (named A, B, C, D, and E for simplicity) in New York, so they decide to lease dedicated lines from the phone company. The phone company will charge for each link made. The costs, in thousands of dollars per year, are shown in the graph.





- In this case, we don't need to find a circuit, or even a specific path; all we need to do is make sure we can make a call from any office to any other. In other words, we need to be sure there is a path from any vertex to any other vertex. If we choose the fewest possible edges from the existing graph that allows it to remain connected, we will be left with a tree. Since this tree will connect all the vertices of the original graph, we can say that it spans the original graph.