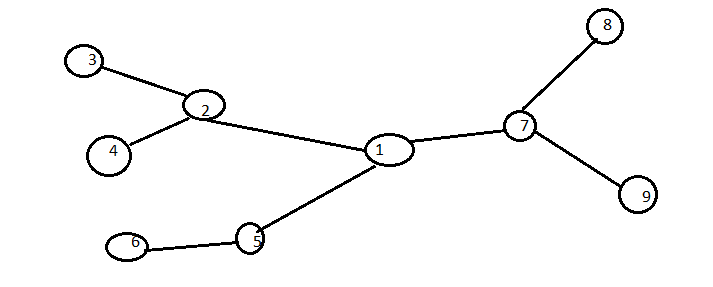
# <http://www.keithschwarz.com/interesting/>

http://ifors.org/tutorial/category/graphs-and-networks/

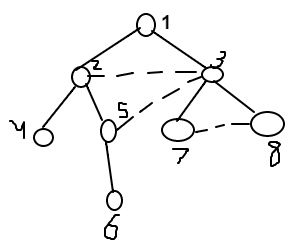
# Graphs

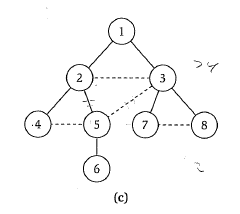
1. **Adjacency Matrix Representation**:
   1. **Space**: V2.
   2. **Use**  for dense graphs where E is close to V2
2. **Adjacency List Representation**:
   1. For sparse graphs with E < V2
   2. Use a linked list or array to represent the edges.
   3. Deleting a node, all linked lists have to be updated.
3. Adjacency Set Representation: Use a Union Find data structure??
4. Graph Representation Comparison

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Space | Checking edge between u and v | Traversal DFS/BFS | Iterate over edges incident to v |
| List of edges | E | E | E+V | E |
| Adj Matrix | V2 | 1 | V2 | V |
| Adj List | V+E | Degree (V) | E+V | Degree(V) |
| Adj Set | E+V | Log(Degree(v)) | E+V | Log(Degree V) |

1. **Directed Graphs**:
   1. Sum of length of adjacency list is E
   2. Edge direction is from u to v
2. Undirected Graphs:
   1. Sum of length of adjacency list is 2E.
   2. Edge direction is from u🡪 v and v🡪u
3. Path of a graph G(V,E) is v1,v2,v3,…vk such that there is an edge between v1 and v2, and v2 and v3… vk-1 to vk.
4. Simple path: Path with no cycles or repeated vertices.
5. Cycle:
   1. A path from v1, v2…vk, for k > 2 such that vk cycles back to v1.
   2. A tree is a graph with no cycles (acyclic directed or undirected graph)
6. **Self Loop**: An edge that connects a vertex to itself.
7. **Parallel Edges**: Two edges are parallel if they connect to the same pair of vertices.
8. **Degree of a vertex**: The number of edges incident to the vertex.
9. **Connected Graph**: For every pair of nodes u and v, there is a path from u to v. In this graph, for every pair of vertex, there is a path (e.g. 3,9)
10. **Non Connected Graph**: A graph that consists of a set of connected components.
11. **Strongly connected graph**: For directed graphs, a directed graph is strongly connected if for every two nodes u and v, there is a path from u to v and from v to u.
12. **Directed Acyclic Graph (DAG)**: A directed graph with no cycles.
13. **Trees**: An undirected graph is a tree if it is connected, and there are no cycles.
14. **Forest**: A disjoint set of trees.
15. **Bipartite Graph**: A graph in which the vertices can be divided in two sets with all vertices connect vertex in one set with the other set.
16. **Weighted Graph**: Weights have been assigned to each edge.
17. **Complete Graph**: Graph with all edges present, i.e. E = n(n-1)/2
18. **Sparse Graph**: A graph where E < VlogV
19. **Range of Edges** can be from 0 to |V|(|V|-1)/2, i.e. every vertex V can be connected to every other vertex (V-1). Divide by two in undirected graph case.
20. An N node tree has N-1 edges
21. For a graph G with N nodes, two of the following imply third:
    1. Graph is connected
    2. Graph has no cycles
    3. Graph has N-1 edges
22. For each j >=1, layer L(i) produced by BFS contains all nodes at a distance exactly j from s.
    1. L0 is the root layer (e.g. 1)
    2. L1 is the one next to it., e.g. 2 , 5, 7
23. Edge Types:
    1. Tree Edge: A new vertex is encountered.
    2. Back Edge: An edge from a descendent vertex to ancestor. From new vertex to a visited vertex?
    3. Forward Edge: Edge from an ancestor to a descendent vertex
    4. Cross Edge: Between trees or subtrees.
24. If there is at most one edge between two vertices, a graph will have (n choose 2) <= n^2 edges.
25. Matrix Space representation is n^2.
    1. Matrix is symmetric for undirected graphs.
26. Adjacency List representation requires O(m+n) space
27. Number of edges in a graph = m = n-1
28. Sum of degrees in a graph = 2m (each edge appears twice)
29. Stack/Queue based traversals (<http://11011110.livejournal.com/279880.html>)
    1. BFS: Mark vertex visited as soon as it is encountered, i.e. in the for loop
    2. DFS: Mark vertex visited when it is scanned for its adjacent edges.
30. TODO: How to find back edges, forward edges and cross edges in directed/undirected graphs.
31. Path of BFS or DFS is in the visited array
32. Parent pointer usage is in the C++ source code.
33. DFS/BFS Applications:
    1. <http://bigfoot.cs.upt.ro/~ioana/algo/Graphs2.ppt>
    2. <http://bigfoot.cs.upt.ro/~ioana/algo/>
34. <http://bigfoot.cs.upt.ro/~ioana/algo/>
35. <http://www.eecs.yorku.ca/course_archive/2012-13/W/2011/Notes/s20_BFS_DFS_apps.pdf>
36. Strongly Connected Components
    1. TODO: Write code for this.
    2. Complexity is O(V+E) as DFS performed on all vertices and vertices are visited at most once.
37. Strongly Connected Components Algorithm:
    1. Do DFS on a graph and calculate finish times on each vertex.
    2. Calculate GT, and do DFS, but do DFS in decreasing order of finish times.
    3. As the vertices are processed, collect vertices from a DFS search into a group. This group forms the connected component.
38. Strongly Connected Components:
    1. The algorithm creates an acyclic component graph GSCC
39. TODO:
    1. Articulation Vertex
    2. Bridge Edge
    3. Euler Tour
    4. Reachability
    5. Cycle Detection
    6. Kruskal Algorithm: Use example with Union Find Algorithm
40. Minimum Spanning Trees
    1. A tree of edges where the weight of the edges is minimized.
41. Kruskal’s Minimum Spanning Tree algorithm
    1. Sorting takes O(ElogE) time.
    2. Number of edges in MST is V-1
    3. Take edge with smallest weight and add it to a UF structure. If successful, add the edge to MST edges. Do this operation for V-1 times
42. Prim’s Algorithm Implementation.
43. Graph Coloring:
    1. <http://www.csun.edu/~danielk/teaching/graph-theory/notes02.pdf>
44. Eulerian Graphs:
    1. <http://www.geeksforgeeks.org/eulerian-path-and-circuit/>

## BFS Traversal

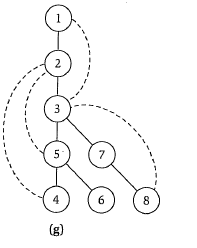
1. BFS Traversal
   1. Page 80-81 Klienberg book
   2. Non Tree Edges: Edges that are in the graph but not in the BFS traversal tree.
2. The non tree edges (x,y) belong to layers that differ by at most 1. They are either within the same layer or one above or one below. 
3. Connected components with BFS (C++ code): Do continuous BFS on a graph till all vertices are visited.
4. BFS Time Complexity:
   1. O(V) for each vertex initialization. Each vertex is enqueued once.
   2. For each vertex, the adjacency list is scanned once.
   3. Sum of length of adjacency list is O(E)
   4. Thus the complexity is O(V+E)
5. **Single Source Shortest Path/Shortest path**: BFS Gives shortest path from a source vertex to a target vertex.
6. BFS Tree of the graph:
   1. Number of vertices in a BFS tree: n
   2. Number of edges in a BFS tree: n-1
   3. Back edges in BFS are not between descendent and ancestor but across nodes.



1. Applications:
   1. Finding all connected components in a graph
   2. Finding all nodes within a connected component.
   3. Finding the shortest path between two nodes.

## DFS Search

1. TODO: DFS Code, iterative and recursive.
2. Complexity:
   1. Adjacency List: O(V+E)
   2. Adjacency Matrix: O(V2)
3. DFS Tree of the graph



1. In a DFS tree, the back edges are between descendent and ancestor nodes. They can cross multiple levels
2. Applications:
   1. Toplogical Sorting
   2. Finding Connected Components
   3. Finding articulation points of a graph
   4. Finding strongly connected components
   5. Solving puzzles such as mazes.

## DFS/BFS Comparison

* DFS has lower memory usage as it does not require storing all of child pointers at each level.
* DFS if looking for a child of a node and the node is lower in the sub-tree.
* If the search vertex is at the top of the tree, BFS will be faster.

|  |  |  |
| --- | --- | --- |
| Applications | DFS | BFS |
| Spanning forest, connected components, paths, cycles | Yes | Yes |
| Shortest Path |  | Yes |
| Minimal usage of memory | Yes |  |

## Topological Sort

1. An ordering of vertices in which each vertex comes before all other vertices in which it has outgoing edges. Used only in DAGs.
2. Properties:
   1. All pairs of consecutive vertices in the sorted order are connected by an edge.
   2. These edges for a Hamiltonian path.
   3. If there are more than one topological sort, then there cannot be a Hamiltonian path?
   4. Topological sort can give cycles if using the indegree=0 method.
3. Algorithm 1 (Using in degree)
   1. Collect all vertices in the graph that have an in degree of 0, i.e. no edges coming into it. Put them in a queue. Increment the counter as well.
   2. While the queue is not empty
      1. Dequeue a vertex, add it to a stack or a linked list, increment a counter
      2. Reduce the indegree of all its adjacent vertices by 1
      3. If one of the vertices has a degree of 0, add it to the queue.
   3. If the size of the collected vertices is not the same as the size of the graph’s vertices, there is a cycle.
4. Topological Sorting:
   1. Do DFS on each vertex, as each vertex is finished, either put it in front of a linked list or push it to a stack.
   2. Assign a finish time to each vertex, so the higher finish time is at top of the stack or at the front of the linked list.
   3. O(V+E) as each vertex is visited only once.
5. Applications:
   1. Course prerequisites.
   2. Detecting deadlocks
   3. Pipeline of computing jobs
   4. Checking for symbolic link loop
   5. Evaluating formulae in spreadsheet.

## Shortest Path Algorithms

##### Variations of shortest path algorithm

* Shortest path in an un-weighted graph
* Shortest path in a weighted graph
* Shortest path in a weighted graph with negative weights.

##### Applications of shortest path algorithm:

* Find fastest way to go from one point to another.
* Find cheapest way to fly/send data

### Shortest path Algorithms

##### Unweighted shortest path

Complexity is (V2) or (V+E)

##### Dijkstra’s algorithm for weighted graphs

Properties:

* Generalization of BFS search.
* BFS cannot give shortest path as it cannot guarantee that the vertex at the front of the queue is the vertex closest to the source vertex.
* Uses the greedy method, always picks the next closest vertex.
* Uses priority queue to store unvisited vertices by distance from s
* Does not work with negative weights.

Difference from un-weighted shortest path

* To represent weights in the adj list, each vertex contains the weight of the edge in addition to their identifier.
* Priority queue used (distance is the priority), and the vertex with the smallest distance is selected for processing.
* Distance to a vertex is calculated by sum of weights of the edges on the path from source to that vertex.
* Update distance in case the newly computed distance is smaller than the old distance.

Code/Algorithm

* Add a vertex to a priority queue.
* Initialize all distances to be -1, distance of vertex (s) to be 0
* For each item in the queue
  + Delete the minimum item (v) in the PQ (distance based).
  + For all children of this item,
    - If D[w] is -1
      * set D[w] = D[v] + weight [v,w]
      * Insert w in the PQ
      * Update path P[w] = v
    - If D[w] is not -1 and > D[v] + weight [v,w]
      * Update D[w] = D[v] + weight[v,w]
      * Update PQ for w with new distance
      * Path[w] = v

Performance

* Depends on number of DeleteMins (V DeleteMins) and updates for priority queue (E updates)
* Binary heap has complexity of O (ElogV), i.e. E updates, each taking logV time.
* If a set is used, then it is: O(E+V2)

Disadvantages

* Does a blind search, cannot handle negative edges.

Relatives

* Bellman-Ford: single source shortest path in a weighted graph. Same concept as Dijkstra’s but can handle negative edges as well. Has a higher running time.
* Prim’s Algorithm: Finds minimum spanning trees in a connected weighted graph.

### Bellman Ford Algorithm

* Complexity (VE)
* Can be applied to directed or undirected graphs.
  + Un directed graphs: Add edge (u🡪v) and (v🡪u) to make it directed.
* An acyclic graph of V vertices contains at most V-1 edges (no cycles)
* Makes (v-1) passes over all edges, relaxing the edges each time.
* Can not contain negative weight cycles or positive cycles but can contain negative weight edges.
* Can be used to detect negative weight cycles

### Topological sort for DAG shortest path method

* Sort the DAG and get topological sort of the vertices.
* Initialize for single source, set the d[s] to 0
* For each vertex in the topological sorted order,
  + For each edge incident to that vertex
  + Relax (u,v,w)

### Shortest path overview

|  |  |
| --- | --- |
| Shortest path in unweighted graph, modified BFS | O(E+V) |
| Shortest path in a weighted graph, Dijkstra | * Array Based (V2+E) * Heap: (V+E)logV * Fibonacci Heap: VlogV+E |
| Shortest path in weighted graph with negative weight [Bellman-Ford] | O(E.V) |
| Shortest path in a weighted acyclic graph (topological sort method) | O(E+V) |

# ALL PAIRS SHORTEST PATH PROBLEMS

#### Floyd Warshell Algorithm

# LONGEST PATH IN A DAG

[Longest path does not have the optimal sub structure](file:///C:\Users\kg\Desktop\interview_files\graphs\longest-path-in-dag.pdf) [[Source](http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/11-Graph/Docs/longest-path-in-dag.pdf)]

# Minimal Spanning Trees

A minimum spanning tree of a graph is a tree formed by edges of the graph such that all vertices are connected, and the weight of the edges is minimal. Example

A graph with 3 spanning trees, one or more can be minimum spanning tree.

Prims Algorithm

Running time is O(V2) without heaps and O(ElogV) for binary heaps.

Kruskal’s Algorithm

## http://www.sanfoundry.com/java-programming-examples-graph-problems-algorithms/

Number of edges in a simple graph with n vertices

There are n(n-1)/2 edges: (n-1) + (n-2) + (n-3) …. 1

First edge can connect to n-1 vertices, second to n-2….

###### How many Different adjacency matrices does a graph of n vertices, E edges have

n! as there are n vertices.

*How many Different adjacency lists does a graph with n vertices have?*

E! i.e. number of permutations of edges.

Graph