Module 5

**

Definition

- A mathematical non-linear data structure
- Capable of representing many kinds of physical structures
- A graph G(V, E) is defined as a set of vertices (V) and a set of edges (E)
- Vertices
- Collection of nodes, represented as points or circles
- Edges
- Connect a pair of vertices and can have weights attached

Examples

- Most unrestricted form of data organization
- Final destination for problem solving
- Many variances

Types

- Undirected graph
- No arrows
- Directed graph
- arrows
- Weighted graph
- Weights on the edges
- Connected graph
- Non-connected graph

Adjacency Matrix Representation

- Create a matrix (e.g. use a 2-dimensional array)
- Any index i represents a node
- Any entry (i,j) in the matrix represents connectivity between the two nodes i,j
- Entry (i,j) = 1 = > an edge exists
- entry(i,j) = 0 => no edge exists

Adjacency List Representation

- Create an array of lists (e.g. use a linked list)

- The headers of the lists represent the nodes
- Every list represents the nodes, connected from the header node

Adjacency Matrix for Directed Graph

Adjacency List for Directed Graph

Insert

Delete

Traversal (Search)

- Depth First Search (DFS)
- Pick a starting point
- Visit this vertex
- Push it onto a stack
- Stack is Last In First Out
- Last pushed item gets popped first
- Mark it
- Rules

1. If possible, visit an adjacent unvisited vertex, mark it, and push it into the stack
2. If can’t follow rule 1, then pop a vertex off the stack
3. If can’t follow rule 1 ot rule 2, you are done

- Idea
- If possible, go (in depth) forward else back track
- Problem
- Since we have cycles, each node may be visited infinite times
- Solution
- Use a boolean visited array
- Breadth First Search (BFS)
- Idea
- Start from the starting node (this will be given to you)
- Generate a traversal tree while traversing the graph
- A node is traversed when its all successor nodes are generated
- Pick a starting point
- Visit this vertex
- Mark it

- Rules

1. Visit the next unvisited vertex (if there is one) that’s adjacent to the current vertex. Mark it and insert it into the queue

- Queue is First In First Out
- First item pushed is first item popped

2. If can’t follow rule 2, because there is no more unvisited vertices, remove a vertex from the queue and mark it the current vertex
3. If you can't carry out rule 2 because the queue is empty, you are done

Spanning tree

- For a weighted undirected graph, spanning tree is a sub-graph that connects all the vertices together using the minimum number of edges required
- The graph should be connected
- There should be no cycles
- For n number of vertices, (n-1) edges are needed
- Hence, for four vertices, we need three edges without any cycles
- Can have multiple spanning trees in a graph
- The minimum spanning tree has the lowest weight
- Minimum Spanning Tree
- Cost of any spanning tree
- Add the weights of all the edges
- Many spanning trees for a single graph
- Calculate cost of all spanning trees and pick the minimum cost spanning tree
- A tree with minimum weights is MST
- Time complexity
- Exponential
- Prim’s Algorithm
- At a glance
- Maintains two sets
- MST set: vertices that are included in the spanning tree
- Set 2: vertices that are not yet included in the spanning tree
- Algorithm
- Key idea
- Find the local optimum in the hopes of finding a global optimum
- Start from one vertex and keep adding edges with the lowest weight until all vertices are reached
- Steps
- Initialize the MST with a vertex chosen at random

- Find al the edges that connect the tree to new vertices, selected the minimum and add it to the tree
- Keep repeating step 2 until we get a minimum spanning tree
- Algorithm

1. Create an array of size V and initialize it with NIL
2. Create a priority queue (min heap) of size V. Let the min heap be Q
3. Insert all vertices into Q. Assign cost of starting vertex to 0 and the cost of other vertices to infinite

- Pseudo code
- Order of matrix selection
- ???
- Use the parent matrix to draw the MST
- Draw all vertices (without the edges)
- Look at the parent to determine the edges
- Use original graph to find weights
- Method
- Extract P
- Extract S
- Extract T or R (we do T)
- Extract R
- Extract Q or U (we do Q)
- Extract U
- Extract V
- Extract W or X (we do X)
- Extract W

Dijkstra’s Algorithm

- Single source shortest path
- Greedy approach
- Always makes local optimal choice at each stage wth the intent of finding a global optimum
- Might not always produce an optimal solution
- Nonetheless, yields local optimal solutions that approximate a global optimal solution in a reasonable amount of time.
- Edge-relaxation approach
- Cannot be used with graphs that have negative weight edges

- Given a graph and a source vertex, find shortest paths from source to all vertices in the given graph
- Graphs must be connected
- Can be directed or non directed
- Similar to MST
- Like MST
- Generate a shortest path tree with given source at root
- Maintain two sets
- One set contains vertices included in the shortest path tree
- Other set includes vertices not yet included in the shortest path tree
- At every step
- We find a vertex which is in set 2 and has a minimum distance from the source
- Outputs value of shortest path
- Not the path itself
- With slightly modification, we can obtain the path too
- Shortest paths
- Main idea
- Relaxing edges in the graph
- Example flow
- For a

Floyd-Warshall Algorithm

- All-source shortest path
- A weighted, directed graph is a collection vertices connected by weighted edges (where the weight is some real number)
- One of the most common examples of a graph in the real world is a road map
- Each location is a vertex, and each road connecting locations is an edge
- We can think of the distance traveled on a road from one location to another as the weight of that edge
- Adjacency matrix
- Let D be an edge-weighted graph in adjacency-matrix form
- D(i,j) is the weight of edge (i,j) or infinity if there is no such edge
- Update matrix D, with the shortest path through immediate vertices
- Dijkstra’s doesn’t work with negative-weight edges
- Dijkstra runs in O(V^3) because it is V^2 and we call it V number of times
- Floy Warshall is same
- Working principle
- Vertices in a graph: numbered one to n
- Consider the subset {1,2,...k} of these n vertices

- Imagine finding the shortest path from vertex i to j using vertices in the set {1,2,...,k} only
- Two situations
- K is an intermediate vertex on the shortest path
- K is not an intermediate vertex on the shortest path
- Example
- D matrix (1 more than number of vertices), a vertex to itself is zero, two vertices with no path is infinity, adjacency matrix; k is intermediate vertex
- D0: draw it as it is
- D1: k = 1, copy and paste first row and first column from D0
- Make diagonals zero
- D(row, col)
- D(3,2) = D(3,1) + D(1,2)
- 2 -> 1, 1 -> 3; 2 is smaller than 8 so keep 2
- 3 -> 1, 1 -> 2, 10 is smaller than infinity so change it to ten
- Mark it
- D2: k = 2, copy and paste second row and second column from D1
- Make diagonals 0
- D(1,3) = D(1,2) + D(2,3); this is 6 which is better than 7
- Mark it
- 3 -> 2, 2 -> 1
- 11
- Keep 6, it is better
- D3: k = 3, copy and paste third row and column from D2
- Make diagonals 0
- Nothing changes
- 2 -> 3, 3 -> 1
- 1 -> 3, 3 -> 2
-
- P matrix
- P0: make grid of vertices, if a slot from D is zero, put NIL, else put vertex row number from D here, infinity will also be NIL
-

	1	2	3
1	NIL	1	1
2	2	NIL	2
3	3	NIL	NIL

- P1:

- Nothing changed in first row, so copy paste from P0
- Nothing changed in second row, so copy paste from P0
- 10 changed in third row
- Copy paste items from P0 in places where no change occurred
- P1(3,2) = P0(1,2)
- P1(i,j) = P0(k,j)
- K = 1
- P0(k,j) = 1
- Fill in one,
-

	1	2	3
1	NIL	1	1
2	2	NIL	2
3	3	1	NIL

- P2:
- P2(1,3) = P1(2,3)
- K = 2
- P1(2,3) = 2
-

	1	2	3
1	NIl	1	2
2	2	NIL	2
3	3	1	NIL

- P3:
- Nothing changed from D2 to D3, so copy P2
-

	1	2	3
1	NIl	1	2
2	2	NIL	2
3	3	1	NIL

- How to reconstruct this path from vertex1 to vertex2 (final p table is given)
-
- First look path path[1][2] = 3
- This signifies that on the path from 1 to 2, 3 is the last vertex visited before 2
- Thus, the path is now 1 … 3 -> 2
- Now look at path[1][3] = 4, thus we find the last vertex visited on the path from 1 to 3 is 4
- The path is now 1 … 4 -> 3 -> 2
- Now we look at path[1][4] = 5. Thus we know our path is
- 1 … 5 -> 4 -> 3 -> 2
- When we look at path[1][5], we find 1, which means our path is
- 1 -> 5 -> 4 -> 3 -> 2

**