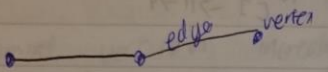
Graphs Note Programming

UNDIRECTED GRAPHS

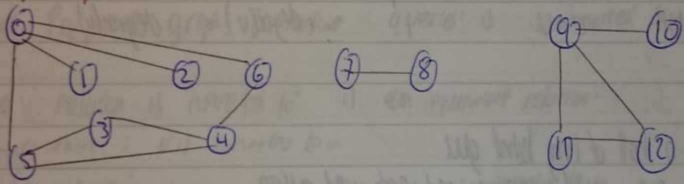
Graph - Set of vertices connected pairwise by edges
Path - Sequence of vertices connected by edges
Cycle: Path whose first and last vertices are the same

Two vertices are connected if there is a path between them



Maintain a list of the edges by linked list or array

Example Graph -



Maintain a V by V boolean Array

For each edge v-w in graph adj [v][w] = adj [w][v] = true

	0	1	2	3	4	5	6	7	8	9	10	11	12
0		1	1			1	1						
1	1												
2	1												
3					1	1							
4				1		1	1						
5	1			1	1								
6	1				1								
7									1				
8								1					
9											1	1	1
10										1	0	0	0
11										1			1
12										1		1	

Maintain vertex-indexed array of lists

adj:

1 [ ] → 6 → 2 → 1 → 5
2 [ ] → 0
3 [ ] → 0
4 [ ] → 5 → 4
5 [ ] → 5 → 6 → 3
6 [ ] → 3 → 4 → 0
7 [ ] → 9 → representation of the same graph
8 [ ] → 7
9 [ ] → 11 → 10 → 12
10 [ ] → 9
11 [ ] → 9 → 12
12 [ ] → 11 → 9

Bag objects

Representation	Space	Add edge	edge between v and w?	Iterate over vertices adj to v?
list of edges	E	1	E	E
adjacency matrix	V^2	1* disallows parallel	1	V
adjacency lists	E+V	1	degree(v)	degree(v)

DFS

- Unroll a ball of string behind you
- Mark each visited intersection and each visited passage
- Retrace steps when no unvisited options
    - Mark v as visited
    - Recursively visit all unmarked vertices w adjacent to v.

```
marked[v] = true
for (int w : G.adj(v)){
   if (!marked[w]){
      dfs(G, w);
      edgeTo[w] = v;
   }
}
```

3

Graphs - Programming

BFS

Puts onto a FIFO queue and mark s as visited
Repeat until queue is empty:
- remove the ~~tea~~ least recently added vertex v
- add each of v's unvisited neighbours to the queue and mark them as visited

BFS examines vertices in increasing distance from s.
Computes shortest path in time proportional to E+V.

Vertices v and w are connected if there is a path between them
Goal: Preprocess graph to answer queries: is v connected to w? in constant time

The relation is "connected to" is ~~an~~ equivalent relation:
- reflexive: v is connected to v
- Symmetric: if v connected to w, w is connected to v
- Transitive: if v connected to w and w connected to x, then v connected to x

Connected components

goal: Partition vertices into connected components
- Initialise all vertices v as unmarked
- For each unmarked vertex v, run DFS to identify all vertices discovered as part of the same component

Digraph - set of vertices connected pairwise by DIRECTED edges
Again maintain vertex indexed array of lists for graph representation
Adjacency list does runtime outdegree(v) for edge from v to w and iterate over vertices pointing from v.

Find all vertices reachable from s along a directed path
- Every undirected graph is a digraph
- DFS is a digraph algorithm

u

Mark v as visited
- recursively visit all unmarked vertices w pointing from v

BFS can also be used

How to implement multi-source constructor for BFS?

Use BFS, but initialise by enqueuing all source vertices

Topological sort.

Goal: Given a set of tasks to be completed with precedence constraints, in which order should we schedule events?

vertex = task, edge = precedence (event before current)

DAG - Directed acyclic graph

Topological sort - redraw DAG so all edges point upwards

(can use DFS)

A digraph has a topological order iff no directed cycle

otherwise runs in circles

Strongly Connected Components

Vertices v and w are strongly connected if there is a directed path from v to w and a directed path from w to v.

Property - v is strongly connected to v
- If v is SC to w, then w is SC to v.
- If v is SC to w and w to x, the v is SC to x

A strong component is a maximal subset of strongly connected vertices

Get post order first, run SCC, if v and w all connected in a cyclic graph they are S.C.

5.

Graph's Programs

Kosaraju's algorithm

- Run DFS on G^R to compute reverse postorder
- Run DFS on G considering vertices in order given by first DFS

MINIMUM SPANNING TREE

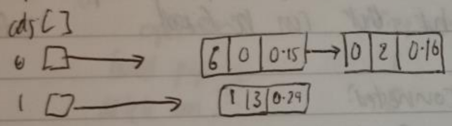
Given - Undirected graph G with positive edge weights (connected)

Def - A spanning tree of G is a subgraph T that is connected and acyclic

Goal - Find a min weight spanning tree

Edge abstraction needed for weighted edges

Representation: maintain vertex-indexed array of Edge lists



Greedy algorithm:

Simplifying assumptions: Edge weights are distinct, graph is connected

Def: A cut in a graph is a partition of it's vertices into two (non-empty) sets. A crossing edge connects a vertex in one set with a vertex in the other.

Cut property: Given any cut, the crossing edge of minimum weight is in the MST.

Proof: Let e be the min weight crossing edge in cut.

- Suppose e is not in MST.
- Adding e to MST creates a cycle
- Some other edge f in cycle must be crossing edge
- Removing f and adding e is also a spanning tree
- Since weight of e is less than the weight of f, that spanning tree is lower weight.
- Contradiction

Greedy Algorithm - start with all edges coloured grey
- Find a cut with no black crossing edges and colour its min-weight edge black
- Continue until V-1 edges are coloured black

Proposition: The greedy algorithm computes the MST.
Proof: Any edge coloured black is in the MST (by cut property).
If fewer than V-1 black edges, there exists a cut with no black crossing edges

What if edge weights are not all distinct?
Greedy MST algorithm still correct if equal weights are present (our correctness proof fails but that can be fixed

What if graph is not connected?
Compute minimum spanning forest MST of each componen

Kruskal's Algorithm for MST.
- Consider edges in ascending order of weight.
- Add the next edge to the tree T unless doing so would create a cycle

Proposition: Kruskal's algorithm computes the MST.
Proof: Kruskal is special case of greedy algorithm
- Suppose Kruskal coloured edge e=v->w black
- Cut = set of vertices connected to v in tree T.
- No crossing edge is black
- No crossing edge has lower weight (because of order)

Challenge: Would adding edge v-w to tree T create a cycle? If not, add it.

Efficient Solution: Use the union find data structure
- Maintain a set for each connected component in T.
- If v and w are in same set, then adding v-w would create cycle.

Programming - Graph

To add v→w, to T, merge sets containing v and w

Create priority queue, add all elements to it.
Create new union find

current = minimum of pq.
get two vertices
if (not connected) add to MST, and union together.

Proposition Kruskal's algorithm computes MST in time proportional to $E \log E$ (in worst case)

Proof:

operation	frequency	time per operation	
build pq	1	E	
delete min	E	$\log E$	
union	V	$\log^* V$	amortized / weighted & compressed
connected	E	$\log^* V$	$\log^* V \le 5$

If edges are already sorted, order of growth is $E \log^* V$

PRIMS MST Algorithm

Start at vertex 0 and greedily grow tree T.
At each step, add to T the min weight edge with exactly one endpoint in T.

Proposition: Prim's algorithm computes the MST.
Proof: Prim is special case of greedy algorithm
- Suppose edge e = min weight edge connecting a vertex on the tree to vertex not on tree
- cut = set of vertices connected on tree
- No crossing edge is black
- No crossing edge has lower weight (edges are ordered)

Use priority queue

8.

① Challenge: Find min weight edge with exactly one endpoint in T.
Lazy Solution: Maintain a PQ of edges with (at least) one endpoint in T.
- key = edge; priority = weight of edge
- Delete min to determine next edge, $e = v \rightarrow w$ to add to T.
- Disregard if both endpoints v and w are in T.
- Otherwise let v be vertex not in T.
  - add to PQ any edge incident to v (assuming endpoint not in T)
  - add v to T.

Proposition: Prim computes MST in time proportional to $E \log E$ and extra space proportional to E (in worst case)

Proof:

Operation	Frequency	binary heap
delete min	E	$\log E$
insert	E	$\log E$

② Challenge: Find min weight edge with exactly one endpoint in T.

Eager Solution: Maintain a PQ of vertices connected by an edge to T, where priority of vertex v = weight of shortest edge connecting v to T. — pq has at most one entry per vertex
- Delete min vertex v and add its associated edge $e = v \rightarrow w$ to T.
- Update PQ by considering all edges $e = v \rightarrow x$, incident to v.
  - ignore if x is already in T.
  - add x to PQ if not already on it.
  - decrease priority of x if $v \rightarrow x$ becomes shortest edge connecting x to T.

Use indexMinPQ: key = edge weight, index = vertex.
(eager version has at most one PQ entry per vertex
Associate an index between 0 and N-1 with each key in a priority queue

a

Programming - Graphs

- Start with same code as MinPQ
- Maintain parallel arrays keys[] pq[] and qp[]:
  - key[i] is priority of i
  - pq[i] is index of the key in heap position
  - qp[i] is the heap position of the key with index i.

WEIGHTED DIRECTED GRAPHS L 26, 27, 28

Shortest path

Which vertices?
- Source-sink: from one vertex to another
- Single source: from one vertex to every other.
- All pairs, between all pairs of vertices

Restrictions on edge weights?
- Non negative weights
- Arbitrary weights
- Euclidean weights

Cycles? No directed cycles
No "negative cycles"

Simplifying assumption: There exists a shortest path from s to each vertex v.

Edge weighted digraph - array list of bags implementation

Goal: find the shortest path from s to every other vertex.
Observation: A shortest paths tree SPT exists
Can represent the SPT with two vertex-indexed arrays.
dist To [v] is length of shortest path from s to v.
edgeTo [v] is last edge on shortest path from s to v.

Edge relaxation

Relax edge $e = v \to w$.

- $distTo[v]$ is length of shortest known path from s to v.
- $distTo[w]$ is length of shortest known path from s to w.
- $EdgeTo[w]$ is last edge on shortest known path from s to w.

If $e = v \to w$ gives shorter path to w through v, update $distTo[w]$ and $edgeTo[w]$

```
int v = e.from,    w = e.to;
if ( distTo[w]  > distTo[v] + e.weight() {
    distTo[w] = distTo[v] + e.weight
    edgeTo[w] = e;
}
```

Proposition: Let G be an edge-weighted digraph.
Then $distTo[\,]$ are the shortest path distance from s iff:

- for each vertex v, $distTo[v]$ is the length of some path from s to v.
- for each edge $e = v \to w$, $distTo[w] \le distTo[v] + e.weight()$.

Proof: Suppose that $distTo[w] > distTo[v] + e.weight()$ for some edge $e = v \to w$. Then e gives a path from s to w (through v) of length less than $distTo[w]$

Suppose that $s = v_0 \to v_1 \to v_2 \to \dots \to v_k = w$ is shortest path from s to w.
Then $distTo[v_k] \le distTo[v_{k-1}] + e_k.weight()$
$distTo[v_{k-1}] \le distTo[v_{k-2}] + e_{k-1}.weight()$
...
$distTo[v_1] \le distTo[v_0] + e_1.weight()$

Add inequalities, and sub $distTo[v_0] = distTo[s] = 0$;
$distTo[w] = distTo[v_k] \le e_k.weight + e_{k-1}.weight + \dots e_1.weight$

Thus $distTo[w]$ is the weight of shortest path to w.

Programming - Graphs

Generic algorithm to compute SPT from s

Initialize distTo[s] = 0 and distTo[v] = ∞ for all other vertices
Repeat until optimality conditions are satisfied:
- Relax any edge

How to choose which edge to relax?
1. - Dijkstra's (non negative weights)
2. - Topological sort (no directed cycles)
3. - Bellman Ford (no negative cycles)

Dijkstra's Algorithm

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest distTo[] value).
- Add vertex to tree and relax all edges pointing from that vertex

Proposition: Dijkstra computes SPT in any edge weighted digraph with non-neg weights
Proof: Each edge e = v→w is relaxed exactly once (when v is relaxed), leaving distTo[w] ≤ distTo[v] + e.weight
- Inequality holds until algorithm terminates because:
  - distTo[w] cannot increase
  - distTo[v] will not change - edge weights are non-neg and we choose lowest distTo[] at each step
- Thus upon termination, SP optimality conditions hold

Insight: Four of our graph search methods are the same algorithm
- maintain a set of explored vertices S
- Grow S by exploring edges with exactly one endpoint leaving S

DFS, BFS, Prim, Dijkstra

12

Acyclic edge-weighted digraphs

It is easier to find shortest paths in edge weight digraph with no directed cycles than a general digraph

Topological: consider vertices in topological order
Relax all edges pointing from vertex

Proposition: Topological sort compute SPT in any edge weighted DAG in time proportion to E+V. (can be negative)

Proof: Edge e = v→w relaxed exactly once, leaving distTo[w] ≤ distTo[v] + e.weight
Inequality holds: distTo[w] cannot increase
distTo[v] will not change

Longest path in edge weighted DAG.

Formulate as a shortest paths problem in edge weight DAG.
- Negate all weights
- Find shortest paths
- Negate weights in result

KEY: Topological sort works with negative edge weights

Parallel job scheduling – Critical path method:

To solve parallel job scheduling problem, create edge weighted DAG
- Source and sink vertices
- Two vertices begin and end for each job
- Source to begin (0 weight)
- end to sink (0 weight)
- One edge for each precedence constraint (0 weight)

Use longest path from the source to schedule each job

13

Programming - Graph

Negative weights

Dijkstra doesn't work
Re-weighting doesn't work

A negative cycle is a directed cycle whose sum of edge weights is negative
A SPT exist exists iff no negative cycles

Bellman-ford algorithm

Initialise distTo[s] = 0 and distTo[v] = ∞ for all other nodes.
Repeat V times: relax each edge

Algorithm	restriction	typical case	worst case	extra space
topological sort	no directed cycles	E+V	E+V	V
Dijkstra (binary heap)	no negative weights	E Log V	E log V	V
Bellman ford	no neg cycles	EV	EV	V
Bellman ford (queue)		E+V	EV	V

- Directed cycle make problem harder
- Negative weight make harder
- Negative cycle make problem intractable

1.

Digraph - set of vertices connected pairwise by directed edges

Problem: Find all vertices reachable from s along a directed path
- every undirected graph is a digraph (with edges in both directions)
- DFS is a digraph algorithm

Mark v as visited.
Recursively visit all unmarked vertices w pointing from v.

BFS (from source vertex s)
→ Put s on FIFO queue and mark s as visited
→ Repeat until the queue is empty:
- remove the least recently added vertex v.
- for each unmarked vertex pointing from v add to Q and mark as visited

Q: How to implement multi-source constructor for find shortest path from several vertexs to one other vertex?

USE BFS but initialize by enqueuing all source vertices

Precedence Scheduling
Given tasks to be completed with prec, what order?
Digraph model: vertex = task edge = precedence constraint

DAG - Directed acyclic graph
Topological sort - redraw DAG so all edges point upwards

Post order go dfs as far as possible → when no further add vertex to start of list → postorder
Reverse postorder is topological sort

Graph has topological order iff no directed cycle

Strongly connected if path $v \to w$ and $w \to v$.

Graphs 1.

V and W are connected if there is a path between them

Goal: pre process graph to answer is v connected to w?

A connected component is a maximal set of connected vertices

→ Initially all vertices v as unmarked

→ for each unmarked vertex v run DFS to identify all vertices discovered as part of the same component

```
{
  marked = new boolean [G.V()];
  id = new int [G.V()];
  for (int v=0; v < G.V(); v++) {
      if (!marked [v]) {
         dfs ( G, v);
         count++;
      }
  }
}
void dfs (G, v) {
     marked [v] = true;
     id[v] = count;
     for (int w : G. adj (v)) {
           if ( ! marked [w])
              dfs (G, w);
           }
     }
}
```

Digraph - set of vertices connected pairwise by directed edges

Digraph Search

Problem: Find all vertices reachable from s along a directed path

- Every undirected graph is a digraph (with edges in both directions)
- DFS is a digraph algorithm
  DFS (to visit a vertex v)
  → Mark v as visited
  → Recursively visit all unmarked vertices w pointing from v.

Same code for digraph as undirected graph for DFS

Multi source shortest paths

Given a digraph and a set of source vertices, find shortest path from any vertex in the set to each other vertex

Implement → Use BFS but initialise by enqueing all source vertices

Topological Sort

Goal: Given a set of tasks to be completed with precedence constraints, in which order should we schedule tasks?

Digraph model: vertex = task, edge = precedence constraint

DAG → Directed acyclic graph
Topological Sort → Redraw DAG so all edges point upwards

Solution: Run dfs. - go to end, record vertex, back step record vertex all the way back to source

This is post order - oldest visited first then youngest at end

Topological order is postorder in reverse

How?

→ Last item in postorder has indegree 0 (no vertices pointing to it.) this is good starting point

→ Second to last can only be pointed to by last item, good follow up

→ 3rd ...

3

Graphs

Proposition: Reverse DFS post order of a DAG is topological order

Proof: Consider any edge v→w, when dfs(v) is called:

① dfs(w) has already been called and returned, thus w was done before v.

② dfs(w) has not yet been called. dfs(w) will get called directly or indirectly by dfs(v) and will finish before dfs(v). Thus w will be done before v.

③ dfs(w) has already been called but has not yet returned. CANT happen in DAG, function call stack contains path from w to v so v→w would complete cycle

Proposition: A digraph has a topological order iff no directed cycle

Proof: If directed cycle → topological impossible

GOAL: Given a digraph, find a directed cycle

Solution: DFS

STRONGLY CONNECTED COMPONENTS

DEF: Vertices v and w are strongly connected if there is a directed path from v to w and a directed path from w to v.

PROPERTY:
- v is S.C to v
- If v S.C to w, then w is S.C. to v.
- If v is S.C. to w, and w to x, v is S.C. to x.

DEF: A strong component is a maximal subset of strongly connected vertices v and w connected if path btw v and w

v and w strongly connected if directed path from v to w and directed from w to v.

~~PRIMS ALGORITHM -LAZY -EAGER MST.~~

~~-Start at vertex 0 and greedily grow tree T.~~

~~-At each step add~~

Minimum Spanning Tree

GIVEN: Undirected graph G with positive edge weights (connected)

DEF: A spanning tree of G is a subgraph T that is connected and acyclic

Goal: Find min weight of spanning tree

Greedy Algorithm

Simplifying assumptions: Edge weights are distinct; graph is connected

DEF: A cut in a graph is a partition of its vertices into two (nonempty) sets.

A crossing edge connects a vertex in one set with a vertex in the other

CUT PROPERTY: Given any cut, the crossing edge of min weight is in MST

PROOF: - Let e be the min-weight crossing edge in cut.
- Suppose e is not in MST.
- Adding e to the MST creates a cycle
- Some other edge f in cycle must be crossing edge
- Since weight e is less than w f, that spanning tree is of lower weight

CONTRADICTION

5.

GRAPHS

Greedy Algorithm
- Start with all edges grey
- Find a cut with no black crossing edges, and color its min weight edge black
- Continue until V-1 edges are black

Proposition: Greedy algorithm computes MST
Proof: - Any edge colored black is in MST (by cut property)
 - If fewer than V-1 black edges, there exists a cut with no black crossing edges

Q: Edges weights not distinct?
 Greedy MST still correct, correctness proof fails but can be fixed

Q. Graph not connected?
 Compute minimum spanning forest = MST of each component

KRUSKAL'S ALGORITHM
- Consider edges in ascending order of weight
- Add next edge to the tree T unless doing so would create a cycle

Prop: Kruskal computes MST
Proof: Kruskal is special case of greedy algo.
 → Suppose kruskal algo colours edge $e = v \to w$ black
 → cut = set of vertices connected to v in tree T.
 - No crossing edge is black
 - No crossing edge has lower weight

CHALLENGE: Would adding $v \to w$ to T create a cycle?
Solution: Use union-find
 - maintain a set for each connected component int.
 - if v and w are in the same set, $v \to w$ will create a cycle

To add v→w, merge sets containing v and w
Using UF : pay in space but gain in time

```
MinPQ <Edge> pq = new MinPQ<Edge>();
for (Edge e: G.edges()) pq.insert(e);

UF uf = new UF (G.V());
while (!pq.isEmpty() && mst.size() < G.V()-1) {
    Edge e = pq.delMin;
    int v = e.either, int w = e.other(v);
    if ( ! uf.connected (v, w)) {
        uf.union(v, w);
        mst.enqueue(e);
    }
}
} //end method
```

Proposition: Kruskal computes MST in time proportional to E log E.

Proof:	freq	time per op
build pq	1	E
delete min	E	log E
union	V	log* V
connected	E	log* V

Using weighted quick union with path compression

If edges are already sorted, order of growth is E log* V.

7.

PRIM ALGORITHM

- Start with vertex 0 and greedily grow T.
- At each step add to T the min weight edge with exactly one endpoint in T.

Proposition: Prim computes MST

Proof: - Prim is special case of greedy:

- Suppose e = min weight edge connecting a vertex on tree to vertex not on tree
- Cut = set of vertices connected on tree
- No crossing edge black
- No crossing edge has lower weight

LAZY SOLUTION:

- Maintain a PQ of edges with (at least) one endpoint in T.
- key = edge, priority = weight of edge
- Delete min to determine next edge e = v→w to add to T.
- Disregard if both endpoints v and w are in T.
- Otherwise let v be vertex not in T
  - add to pq any edge incident to v
  - add v to T.

```
while (! pq.isEmpty()) {
    Edge e = pq.delMin();
    int v = e.either(), int w = e.other(v)
    if (marked[v] && marked[w]) do nothing
    else   mst.enqueue(e);
           if (!marked[v]) visit(G,v)
           if (!marked[w]) visit(G,w)
}
```

8.

```
visit (graph G, int v) {
  marked [v] = true;
  for (edge e : G.adj(v)) {
      if (!marked [e.other(v)])
         pq.insert(e)
      }
  }
}
```

Proposition: LAZY prim computes MST in ElogE and extra space proportional to E in worst case

Proof: Operation | freq | binary heap
delete min | E | log E
insert | E | log E

Eager solution:
- Maintain a pq of vertices connected by an edge to T, where priority of v = weight of shortest edge connecting v to T.
- Delete min vertex v and add associated edge e=v→w to T
  → Update pq by considering all edges e = v→x incident to v.
    → ignore if x is already in T
    → add x to pq if not already on it
    → decrease priority of x if v→x becomes shortest edge ~~adding~~ connecting x to T.

	verlex	distance to vertex from 0.	last edge on path to vertex.
Q E iii.	v	distTo[]	edgeTo[]
	0	0.0	—
	1	5.0	0→1
	2	3.0	0→2
	3		
	4		
	5		
	6		
	7.		

Start now from 2 as shortest distance from 0

V	D	E
0	0.0 ✓	
1	~~5.0~~ 4.0 ✓	~~0→1~~ 2→1
2	3.0 ✓	0→2
3	6 ✓	2→3
4	12 ✓	1→4
5	7 ✓	3→5
6	~~11~~ 9 ✓	~~2→6~~ 5→6.
7.		

2→1 = 3+1 = 4 add

2→3 = 3+3 = 6

2→6 = 3+8 = 11

1→4 12.v.

1→3. 11 x

3→5. 6+1 = 7.

5→4 7+2 = 9 x.

5→6 7+2 = 9