Reference Sheet for CO142.2 Discrete Mathematics II

Spring 2017

1 Graphs

Defintions

1. *Graph*: set of N nodes and A arcs such that each $a \in A$ is associated with an unordered pair of nodes.
2. *Simple graph*: no parallel arcs and no loops.
3. *Degree*: number of arcs incident on given node.
    (a) Sum of degrees of all nodes in a graph is even.
    (b) Number of nodes with odd degree is even.
4. *Subgraph*: G_S is a subgraph of G iff $\text{nodes}(G_S) \subseteq \text{nodes}(G)$ and $\text{arcs}(G_S) \subseteq \text{arcs}(G)$.
5. *Full (induced) subgraph*: contains all arcs of G which join nodes present in G_S.
6. *Spanning subgraph*: $\text{nodes}(G_S) = \text{nodes}(G)$.

Representation

1. *Adjacency matrix*: symmetric matrix with number of arcs between nodes, count each loop twice.
2. *Adjacency list*: array of linked lists, if multiple arcs then multiple entries, count loops once.
3. An adjacency matrix has size n^2, an adjacency list has size $\leq n + 2a$ (better for sparse graphs).

Isomorphisms

1. *Isomorphism*: bijection f on nodes together with bijection g on arcs such that each arc a between nodes n_1 and n_2 maps to the node $g(a)$ between nodes $f(n_1)$ and $f(n_2)$. Same adjacency matrix (possibly reordered).
2. *Automorphism*: isomorphism from a graph to itself.

Testing for isomorphisms:

1. Check number of nodes, arcs and loops. Check degrees of nodes.
2. Attempt to find a bijection on nodes. Check using adjacency matrices.

Finding the number of automorphisms: find number of possibilities for each node in turn. Then multiply together.

Planar Graphs

1. *Planar*: can be drawn so that no arcs cross.
2. *Kuratowski's thm*: a graph is planar iff it does not contain a subgraph homeomorphic to K_5 or $K_{3,3}$. *Homeomorphic*: can be obtained from the graph by replacing the arc $x - y$ by $x - z - y$ where required.
3. For a planar graph, $F = A - N + 2$.

Graph Colouring

1. *k-colourable*: nodes can be coloured using no more than k colours.
2. *Four colour thm*: every simple planar graph can be coloured with at most four colours.
3. *Bipartite*: nodes can be partitioned into two sets such that no two nodes from the same set are joined. Bipartite is equivalent to 2-colourable.

Paths and Connectedness

1. *Path*: sequence of adjacent arcs.
2. *Connected*: there is a path joining any two nodes.
3. $x \sim y$ iff there is a path from x to y. The equivalence classes of $\sim$ are the *connected components*.
4. *Cycle*: path which finishes where it starts, has at least one arc, does not use the same arc twice.
5. *Acyclic*: graph with no cycles.
6. *Euler path:* path which uses each arc exactly once. Exists iff number of odd nodes is 0 or 2.
7. *Euler circuit:* cycle which uses each arc exactly once. Exists iff every node is even.
8. *Hamiltonian path*: path which visits every node exactly once.
9. *Hamiltonian circuit*: HP which returns to start node.

Trees

1. *Tree*: acyclic, connected, rooted (has distinguished / root node) graph.
   (a) Exists a unique non-repeating path between any two nodes.
   (b) A tree with n nodes has $n-1$ arcs.
2. *Depth*: distance of node (along unique path) from root.
3. *Spanning tree*: spanning subgraph which is a non-rooted tree (lowest-cost network which still connects the nodes). All connected graphs have spanning trees, not necessarily unique.
4. *Minimum spanning tree*: spanning tree, no other spanning tree has smaller weight. *Weight* of spanning tree is sum of weights of arcs.

Directed Graphs

1. *Graph*: set of N nodes and A arcs such that each $a \in A$ is associated with an *ordered* pair of nodes.
2. *Indegree*: number of arcs entering given node.
3. *Outdegree*: number of arcs leaving given node.
4. Sum of indegrees = sum of outdegrees = number of arcs.

Weighted Graphs *Weighted graph*: simple graph G together with a weight function $W : \text{arcs}\,(G) \to \mathbb{R}^+$.

2 Graph Algorithms

2.1 Traversing a Graph

2.1.1 Depth First Search

```
#### Depth First Search ####

visited[x] = true
print x
for y in adj[x]:          # Follow first available arc
  if not visited[y]:
    parent[y] = x
    dfs(y)                # Repeat from the discovered node
                          # Once all arcs followed, backtrack
```

2.1.2 Breadth First Search

```
#### Breadth First Search ####

visited[x] = true
print x
enqueue(x,Q)
while not isEmpty(Q):
  y = front(Q)
  for z in adj[y]:          # Follow all available arcs
    if not visited[z]:
      visited[z] = true
      print z
      parent[z] = y
         enqueue (z,Q)      # Repeat for each discovered node
  dequeue(Q)
```

For both methods, each node is processed once, each adjacency is list processed once, giving running time of $O\,(n + a)$.

Applications

1. Find if a graph is connected.
2. Find if a graph is cyclic: use DFS, if we find a node that we have already visited (except by backtracking) there is a cycle.
3. Find distance from start node: BFS with a counter.

2.2 Finding Minimum Spanning Trees

2.2.1 Prim's Algorithm

```
#### Prim's Algorithm ####

                                    # Choose any node as start
tree[start] = true                  # Initialise tree as start
for x in adj[start]:                # and fringe as adjacent nodes
  fringe[x] = true
  parent[x] = start
  weight[x] = W[start,x]
while fringe nonempty:
  f = argmin(weight)                # Select node f from fringe
                                    # s.t. weight is minimum
  tree[f] = true                    # Add f to tree
  fringe[f] = false                 # and remove from fringe
  for y in adj[f]:
    if not tree[y]:
      if fringe[y]:                 # Update min weights
                                    # for nodes already in fringe
        if W[f,y] < weight[y]:
          weight[y] = W[f,y]
          parent[y] = f
      else:                         # and add any unseen nodes
                                    # adjacent to f to fringe
        fringe[y] = true
        weight[y] = W[f,y]
        parent[y] = f
```

Method takes $O\left(n^2\right)$ time. For each node, we check at most n nodes to find f.

Correctness of Prim's *Inductive step*: Assume that $T_k \subseteq T'$ where T' is some MST of a graph G. Suppose we choose $a_{k+1} \notin \text{arcs}\left(T'\right)$, between nodes x and y. There must be another path from x to y. So this path uses another arc a that crosses the fringe - but $W\left(a_{k+1}\right) \leq W\left(a\right)$ so $T_{k+1} \subseteq T'$.

Implementation of Prim's with Priority Queues Using binary heap implementation, operations are $O\left(\log n\right)$ except isEmpty and getMin which are $O\left(1\right)$. This gives $O\left(m \log n\right)$ overall - better for sparse graphs.

2.2.2 Kruskal's Algorithm

```
#### Kruskal's Algorithm ####

                          # Assign nodes of G numbers 1..n
Q = ...                   # Build a PQ of arcs with weights as keys
sets = UFcreate(n)        # Initialise Union-Find with singletons
F = {}                    # Initialise empty forest
while not isEmpty(Q):
  (x,y) = getMin(Q)       # Choose the arc of least weight
  deleteMin(Q)
  x' = find(sets,x)       # Find which components it connects
  y' = find(sets,y)
  if x' != y':            # If components are different/no cycle made
    add (x,y) to F        # add arc to forest
    union(sets,x',y')     # and merge two components
```

Implementation of Kruskal's with Non-binary Trees

1. Each set stored as non-binary tree, where root node represents leader of set.
2. Merge sets by appending one tree to another. We want to limit depth, so always append tree of lower size to one of greater size.
3. Depth of a tree of size k is then $\leq \lfloor \log k \rfloor$.

Gives $O\left(m \log m\right)$ which, since m is bounded by n^2, is equivalent to $O\left(m \log n\right)$. Better for sparse graphs.

Improvements for Kruskal's with Path Compression

1. When finding root/leader of component containing the node x, if it is not parent$\left[x\right]$, then we make parent$\left[y\right] = $ root for all y on the path from x to the root.
2. This gives $O\left(\left(n+m\right) \log^* n\right)$ where $\log^* n$ is a very slowly growing function.

2.3 Shortest Path Problem

2.3.1 Dijkstra's Algorithm

```
#### Dijkstra's Algorithm ####

tree[start] = true                    # Init tree as start node
for x in adj[start]:                  # Init fringe as adj nodes
  fringe[x] = true
  parent[x] = start
  distance[x] = W[start,x]
while not tree[finish] and fringe nonempty:
  f = argmin(distance)                # Choose node with min distance
  fringe[f] = false                   # remove from fringe
```

```
  tree[f] = true                    # and add to tree
  for y in adj[f]:                  # For adj nodes
    if not tree[y]:                 # if in fringe, update distance
      if fringe[y]:
        if distance[f] + W[f,y] < distance[y]:
          distance[y] = distance[f] + W[f,y]
          parent[y] = f
      else:                         # otherwise, add to fringe
        fringe[y] = true
        distance[y] = distance[f] + W[f,y]
        parent[y] = f
return distance[finish]
```

Requires $O\left(n^2\right)$ steps (or $O\left(m \log n\right)$ with PQ).

Correctness Prove by the following invariant:

1. If x is a tree or fringe node, then parent $[x]$ is a tree node.
2. If x is a tree node, then distance $[x]$ is the length of the shortest path and parent $[x]$ is its predecessor along that path.
3. If f is a fringe node then distance $[f]$ is the length of the shortest path where all nodes except f are tree nodes.

2.3.2 Floyd's Algorithm

```
#### Floyd's Algorithm ####

A                                 # Input adj matrix
B[i,j]
  if i = j      : 0
  if A[i,j] > 0: A[i,j]
  otherwise     : INFINITY        # Init shortest paths via no nodes
for k = 1 to n:
  for i = 1 to n:
    for j = 1 to n:
      B[i,j] = min(B[i,j],        # New shortest path is direct
               B[i,k] + B[k,j])   # or concat shortest paths via k
return B
```

Warshall's algorithm is the same but uses Boolean values (determines whether a path exists). Both have complexity $O\left(n^3\right)$.

Correctness We discuss the *inductive step*: Suppose there is a shortest path p from i to j using nodes with identifiers $< k$, of length d. Either:

1. k is not an intermediate node of p. Then $B_{k-1}[i,j] = d$ already.
2. k is an intermediate node of p. Then $B_{k-1}[i,j] = B_{k-1}[i,k] + B_{k-1}[k,j]$ since these give shortest paths to and from k.

Dynamic Programming Both algorithms are examples of dynamic programming:

1. Break down the main problem into sub-problems.
2. Sub-problems are ordered and culminate in the main problem.

2.4 The Travelling Salesman Problem

Problem Given a *complete* weighted graph, find a minimum weight tour of the graph visiting each node exactly once.

2.4.1 Bellman-Held-Karp Algorithm

```
#### Bellman-Held-Karp Algorithm ####

start = ...                                # Choose some start node
for x in Nodes \ {start}:                  # Set costs on empty set
  C[{},x] = W[start,x]                     # as direct weight from start
for S in subsets(Nodes \ {start}):         # For sets S of increasing size
  for x in Nodes \ (S union {start}):      # for each node x out of set
    c[S,x] = INFINITY
    for y in S:
      C[S,x]                               # update the cost to minimum
        = min(C[S \ {y},y] + W[y,x],       # that visits all nodes in S
          C[S,x])                          # and then x
opt = INFINITY
for x in Nodes \ {start}:                  # Then choose the min value
  opt = min(C[Nodes \ {start,x},x]         # for an S without x plus
        + W[x,start],opt)                  # weight of x from start
return opt
```

Dynamic programming approach that requires $O\left(n^2 2^n\right)$ steps.

2.4.2 Nearest Neighbour Algorithm

Always choose the the shortest available arc. Not optimal but has $O\left(n^2\right)$ complexity.

3 Algorithm Analysis

3.1 Searching Algorithms

3.1.1 Searching an Unordered List L

For an element x:

1. If x in L return k s.t. $L[k] = x$.
2. Otherwise return "not found".

Optimal Algorithm: Linear Search Inspect elements in turn. With input size n we have time complexity of n in worst case.

Linear search is optimal. Consider any algorithm A which solves the search problem:

1. *Claim:* if A returns "not found" then it must have inspected every entry of L.
2. *Proof:* suppose for contradiction that A did not inspect some $L[k]$. On an input L' where $L'[k] = x$, A will return "not found", which is wrong. So in worst case n comparisons are needed.

3.1.2 Searching an Ordered List L

Optimal Algorithm: Binary Search Where $W(n)$ is the number of inspections required for a list of length n, $W(1) = 1$ and $W(n) = 1 + W(\lfloor n/2 \rfloor)$. Gives $W(n) = 1 + \lfloor \log_2 n \rfloor$.

1. *Proposition*: if a binary tree has depth d, then it has $\leq 2^{d+1} - 1$ nodes.
2. *Base Case*: depth 0 has less than or equal to $2 - 1 = 1$ node. True.
3. *Inductive Step*: assume true for d. Suppose tree has depth $d + 1$. Children have depth $\leq d$ and so $\leq 2^{d+1} - 1$ nodes. Total number of nodes $\leq 1 + 2 \times \left(2^{d+1} - 1\right) = 2^{(d+1)+1} - 1$.
4. *Claim*: any algorithm A must do as many comparisons as binary search.
5. *Proof*: the tree for algorithm A has n nodes. If depth is d then $n \leq 2^{d+1} - 1$. Hence $d + 1 \geq \lceil \log(n+1) \rceil$. For binary search, $W(n) = 1 + \lfloor \log n \rfloor$ which is equivalent to $\lceil \log(n+1) \rceil$.

3.2 Orders

1. f is $O(g)$ if $\exists m \in \mathbb{N} \exists c \in \mathbb{R}^+ . [\forall n \geq m . (f(n) \leq c \times g(n))]$.
2. f is $\Theta(g)$ iff f is $O(g)$ and g is $O(f)$.

3.3 Divide and Conquer

1. Divide problem into b subproblems of size n/c.
2. Solve each subproblem recursively.
3. Combine to get the result.

E.g. Strassen's Algorithm for Matrix Multiplication

1. Add extra row and or column to keep dimension even.
2. Divide up matrices into four quadrants, each $n/2 \times n/2$.
3. Compute each quadrant result with just 7 multiplications (instead of 8).

3.4 Sorting Algorithms

3.4.1 Insertion Sort

1. Insert $L[i]$ into $L[0..i-1]$ in the correct position. Then $L[0..i]$ is sorted.
2. This takes between 1 and i comparisons.
3. *Worst case*: $W(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ (occurs when in reverse order). Order $\Theta\left(n^2\right)$.

3.4.2 Merge Sort

1. Divide rougly in two.
2. Sort each half seperately (by recursion).
3. Merge two halves.
4. *Worst case*: $W(n) = n - 1 + W\left(\left\lceil \frac{n}{2} \right\rceil\right) + W\left(\left\lfloor \frac{n}{2} \right\rfloor\right) = n\log(n) - n + 1$ assuming n can be written as 2^k.

3.4.3 Quick Sort

1. Split around the first element.
2. Sort the two sides recursively.
3. *Worst case*: $W(n) = n - 1 + W(n-1) = \frac{n(n-1)}{2}$ (occurs when already sorted).
4. *Average case*: $A(n) = n - 1 + \frac{1}{n}\sum_{s=1}^{n}(A(s-1) + A(n-s)) = n - 1 + \frac{2}{n}\sum_{i=2}^{n-1}A(i)$, asuming each position s is equally likely. $A(n)$ is order $\Theta(n\log n)$.

3.4.4 Heap Sort

1. Add elements to a priority queue, implemented with binary heap. Read off the queue.
2. Is $\Theta(n\log n)$ - also can be performed in place.
   (a) $\Theta(n)$ inserts each taking $\Theta(\log n)$.
   (b) $\Theta(n)$ gets each taking $\Theta(1)$.
   (c) $\Theta(n)$ deletions each taking $\Theta(\log n)$.

3.4.5 Parallel Sorting

1. Merge sort can be parallelised by executing recursive calls in parallel.
2. Work still same but time taken reduced.
3. *Worst case (time taken)*: $W'(n) = n - 1 + W'\left(\frac{n}{2}\right) = 2n - 2 - \log n$.

3.4.6 Odd / Even Merge

1. To merge L_1 and L_2:
   (a) Take odd positions and merge to get L_3 ($O(\log n)$).
   (b) Take even positions and merge to get L_4 ($O(\log n)$).
   (c) Do an interleaving merge on L_3 and L_4 ($O(1)$).
2. Merge is $O(\log n)$ instead of $O(n)$ (sequential).
3. Exploits odd/even networks (made up of comparators).
4. Time taken is now $(\log n)(1 + \log n)/2$ which is $\Theta\left(\log^2 n\right)$.

3.4.7 Lower Bounds

We express the sorting algorithm as a *decision tree*. Internal nodes are *comparisons*. Leaves are *results*. The worst case number of comparisons is given by depth.

Minimising Depth (Worst Case)

1. There are $n!$ permutations so we require $n!$ leaves.
2. *Proposition*: if a binary tree had depth d then it has $\leq 2^d$ leaves.
3. *Proof*: simple proof by induction.
4. *Lower bound (worst case)*: $2^d \geq n!$ so $d \geq \lceil \log(n!) \rceil$.
5. $\log(n!) = \sum_1^n \log(k) \approx \int_1^n \log x \, \mathrm{d}x = n\log(n) - n + 1$.

Minimising Total Path Length (Average Case)

1. *Balanced tree*: tree at which every leaf is at depth d or $d-1$.
2. *Proposition*: if a tree is unbalanced then we can find a balanced tree with the same number of leaves without increasing total path length.
3. *Lower bound (average case)*: must peform at least $\lfloor \log(n!) \rfloor$ calculations (since total path length is minimum for balanced trees).

3.5 Master Theorem

For $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, with the critical exponent $E = \frac{\log a}{\log b}$:

1. If $n^{E+\epsilon} = O(f(n))$ for some $\epsilon > 0$ then $T(n) = \Theta(f(n))$.
2. If $f(n) = \Theta\left(n^E\right)$ then $T(n) = \Theta(f(n)\log n)$.
3. If $f(n) = O\left(n^{E-\epsilon}\right)$ for some $\epsilon > 0$ then $T(n) = \Theta\left(n^E\right)$.

4 Complexity

4.1 Tractable Problems and P

P: Class of decision problems that can be easily *solved*.

1. *Tractable problem*: efficiently computable, focusing on worst case.
2. *Decision problem*: Has a yes or no answer.
3. *Cool-Karp Thesis*: Problem is tractable if it can be computed within polynomially many steps in the worst case.
4. *Polynomial Invariance Thesis*: If a problem can be solved in p-time in some reasonable model of computation, then it can be solved in p-time in any other reasonable model.
5. P: A decision problem $D(x)$ is in P if it can be decided within time $p(n)$ in some reasonable model of computation, where n is the size of the input, $|x|$.

Unreasonable Models

1. *Superpolynomial Parallelism*: I.e. take out more than polynomially many operations in parallel in a single step.
2. *Unary numbers*: Gives input size exponentially larger than binary.

Polynomial Time Functions

1. *Arithmetical operations* are p-time.
2. If f is a p-time function, then its *ouput size* is polynomially bounded in the *input size*. I.e. $|f(x)| \leq p(|x|)$ - because we only have p-time to build the output.
3. *Composition*: If functions f and g are p-time, then $g \circ f$ is computable. Since f takes $p(n)$ and g then takes $q(p'(n))$ where $p'(n)$ is the output size of f.

4.2 NP

NP: Class of decision problems that can be easily *verified*.

4.2.1 HamPath Problem

Given a graph G and a list p, is p a Ham path of G?

1. Verification of HamPath is in P. HamPath (G) iff $\exists p$Ver-HamPath (G, p).
2. $D(x)$ is in NP if there is a problem $E(x, y)$ in P and a polynomial $p(n)$ such that:
   (a) $D(x)$ iff $\exists y.E(x, y)$.
   (b) If $E(x, y)$ then $|y| \leq p(|x|)$ (E is poly bounded).

4.2.2 Satisfiability Problem

Given a propositional formula ϕ in CNF, is it satisfiable?

1. SAT is not decidable in p-time: we have to try all possible truth assignments - for m variables this is 2^m assignments.
2. SAT is in NP: Guess an assignment v and verify in p-time that v satisfies ϕ.

4.2.3 P⊆NP

1. Suppose that D is in P.
2. To verify $D(x)$ holds, we don't need to guess a certificate y - we just decide $D(x)$ directly.
3. Formally: Define $E(x, y)$ iff $D(x)$ and y is the empty string. Then clearly $D(x)$ iff $\exists y.E(x, y)$ and $|y| \leq p(|x|)$.

4.2.4 P=NP

Unknown.

4.3 Problem Reduction

1. *Many-one reduction*: Consider two decision problems D and D'. D many-one reduces to D' ($D \leq D'$) if there is a p-time function f s.t. $D(x)$ iff $D'(f(x))$.
2. Reduction is reflexive and transitive.
3. If both $D \leq D'$ and $D' \leq D$, then $D \sim D'$.

4.3.1 P and Reduction

1. Suppose algorithm A' decides D' in time $p'(n)$. We define algorithm A to solve D by first computing $f(x)$ and then running $A'(f(x))$.
   (a) Step 1 takes $p(n)$ steps.
   (b) Note that $|f(x)| \leq q(n)$ for some poly q. Step 2 takes $p'(q(n))$.
2. Hence: If $D \leq D'$ and $D' \in \text{P}$, then $D \in \text{P}$.

4.3.2 NP and Reduction

Assumption

1. Assume $D \leq D'$ and $D' \in \text{NP}$.
2. Then $D(x)$ iff $D'(f(x))$.
3. Also there is $E'(x, y) \in \text{P}$ s.t. $D'(x)$ iff $\exists y.E'(x, y)$.
4. Also if $E'(x, y)$ then $|y| \leq p'(|x|)$.

Proof Part A

1. $D(x)$ iff $\exists y.E'(f(x), y)$.
2. Define then $E(x, y)$ iff $E'(f(x), y)$. We can now prove part (a).

Proof Part B

1. Suppose $E(x, y)$. Then $E'(f(x), y)$, hence $|y| \leq p'(|x|)$.
2. $f|x| < q(n)$ for some poly q. So $|y| \leq p'(q(|x|))$, proving part (b).

4.4 NP-Completeness

1. D is NP-*hard* if for all problems $D' \in \text{NP}$, $D' \leq D$.
2. D is NP-*complete* if for all problems $D \in \text{NP}$ and D is NP-hard.

Cook-Levin Theorem SAT is NP-complete.

Intractability

1. Suppose $\text{P} \neq \text{NP}$ and D is NP-hard.
2. Suppose for a contradiction that $D \in \text{P}$.
3. Consider any other $D' \in \text{NP}$. $D' \leq D$ since D is NP-hard. Hence $D \in \text{P}$.
4. Hence $\text{NP} \subseteq \text{P}$. But $\text{P} \subseteq \text{NP}$. Hence $\text{P} = \text{NP}$, which is a contradiction.

Proving NP-Completeness E.g. Prove TSP is NP-complete.

1. Prove TSP is NP (easy).
2. Reduce HamPath (a known NP-hard problem) to TSP.