180 Final Revision

Reminder:

*** **edge cases** !!!
*** give **runtime** !!!
*** O(log n) —> binary search / divide and conquer
*** pairwise relation between objects —> graph
*** brute force O(n^2) —> Dynamic Programming

Bipartite Graph / 2 Colorability / BFS

Sample Question:

We have ‘n’ samples of DNA, each belonging to one of the 2 species A and B. We would like to divide the ‘n’ samples into 2 groups - those that belong to A and that belong to B, but it’s very hard to identify the true species of the individual samples. So, we use the following approach: For each pair of samples ‘i’ and ‘j’, we carefully study them side by side. If we’re confident enough in our judgment, we label them either ‘similar’ or ‘different’. Those pairs for which we are not confident enough are labeled ‘ambiguous’. So now we have a collection of ‘n’ samples, as well as a collection of ’m’ judgments (either ‘same’ or ‘different’) for the pairs that were not labeled ‘ambiguous’. We’d like to know if this data is consistent with the idea that each sample is from one of species ‘A’ or ‘B’. More concretely, we’ll declare the ‘m’ judgments to be consistent if it is possible to label each sample as either ‘A’ or ‘B’ in such a way that for each pair (i,j) labeled ‘same’, they have the same final label; and for each pair (i,j) labeled ‘different’, they have different labels. Can you come up with an O(m+n) algorithm that determines whether a given set of ‘m’ judgments are consistent?!

Answer:

Use 2 colorability:

- choose a vertex u and put it in set A
- for all the “different” judgments m(u,v) that contains u, put v in set B
- for all the “same” judgments m(u,t) that contains u, put t in set A
- if there’s any vertex that exist in both A and B, the graph is not 2 colourable and return
- repeat line 1-4 for other vertices until all m judgments are used
- after all m judgments are used and there’s no any vertex exist in both sets, the nodes are 2 colourable

Determine:
If bipartite (2 colourable) —> consistent
Else if not bipartite (not 2 colourable) —> inconsistent

Runtime:
Worst case scenario, we will have to loop through all vertices to use up the m judgements.
O(m+n)

BFS takes O(m+n)

NOTE:
BFS checks for CYCLES in a graph
Bipartite == cannot contain odd length cycle
BFS can determine if G is **strongly connected** in O(m+n) time

BFS Algorithm
BFS(G, s)
1. set L0 <— {s}
2. initialise LEV(s) <— 0 and LEV(v) <— infinity for all other v
3. for i <— 1 to n
    1. set Li <— {}
    2. for each v in Li-1
        1. for each edge (v, w) in E adjoining v
        2. if LEV(w) > i-1, set LEV(w) <— i and Li <— Li union {w}

DFS
DFS Algorithm
//while == undiscovered //grey == discovered, unfinished //black == finished
DFS(G, s)
1. set color[v] <— white for all v in G
2. set color[s] <— grey and current <—s
3. while current has adjacent non-black nodes:
    1. if current has some adjacent while node x:
        1. set pred[x] <— current //predecessor of x
        2. set current <— x
        3. set color[current] <— grey
    2. else current has no adjacent while node:
        1. set color[current] <— black
        2. set current <— pred[current]

DAG and Topological Order
NOTE:
G has a topological order == G is a DAG == G has **at least** a node with no incoming edges
Cycles cannot be topologically sorted —> the first vertex must have no incoming edges

Topological sorting Algorithm
1. compute the in-degree and out-degree for every vertex
2. all the vertices that have an in-degree of 0 == source. go through the lists and put sources into a separate source list
3. pick one source and output it. Remove it from both lists and from its neighbour's in-degree vertices. If there new vertices with 0 in-degree, put them in list.
4. repeat step 2.

Pessimistic Runtime: O(n^2) // for each source O(n) * modify its neighbour's index O(n)
But better: O(E+n) // only only check the same edge and source one time

Shortest Path Problem / Dijkstra's Algorithm

Dijkstra's Algorithm //*only for positive weighted graph**
Dijkstra (G, 1, s)

1. set d[s] <— 0 and d[v] <— infinity for all v != s
2. set p[v] <— null for all v //precedence
3. set X <— V //unfinished set of vertices
4. while X is nonempty
    1. set x <— the element of X with minimum d[x]
    2. for all edge (x, y)
        1. if d[y] > d[x] + l(x,y) then
            1. set d[y] <— d[x] + l(x, y)
            2. set p[y] <— x
    3. set X <— X - {x}

Runtime:

MST: Kruskal’s / Reverse-Delete / Prim’s [greedy]

Sample Question:

Given a undirected, connected, positive-weighted graph G=(V,E), we all know what a spanning tree of this graph is. Given a spanning tree T of G, we define a bottleneck edge to be an edge of T with the greatest weight/cost. A spanning tree T of G is a minimum-bottleneck spanning tree if there is no spanning tree T of G with a cheaper bottleneck edge.
(a) Is every minimum-bottleneck spanning tree of a graph G a minimum spanning tree of G? Prove or give a counterexample.
(b) Is every minimum spanning tree of G a minimum bottleneck spanning tree of G? Prove or give a counter-example.

Answer:

(a)
Counter-example:
(v1, v2, v3, v4) with edges between every two vertices and value of i+j.

(b)
Yes.
If there exist a “better” MST with larger bottleneck value, we can break the vertices into two sets by not using that bottleneck. There exist a cheaper way to connect the two sets since all the edges in our MST is smaller than the bottleneck edge from “better” MST.
The “better” MST is not the best —> contradiction!!

NOTE:

MST may not be unique
for prim’s and kruskal’s, graph must be **Connected, Weighted**
Cut property: min edge between two sets of vertices must be contained in MST
Cycle property: the max cost edge in a cycle must NOT be contained in MST
Cycle and cutest intersect in an even number of edges
Clustering of Maximum spacing == finding an MST and deleting the k-1 most expensive edges

Prim’s Algorithm //add min cost edge in cutset (expand as one)
Prim(G, c)

1. for each v in V set a[v] <— infinity
2. initialise an empty priority queue Q
3. for each v in V insert v onto Q
4. initialise set of explored nodes S <— null
5. while (Q is not empty)
   1. u <— delete min element from Q
   2. S <— S union {u}
   3. foreach (edge e = (u, v) incident to u)
      1. if ((v is not in S) and (cost of e < a[v]))
         1. decrease priority a[v] to cost of e

Runtime: O(m*logn)

Kruskal's Algorithm //add min cost edge in cutset (expand as multiple groups)
Kruskal(G, c)
1. sort edges weights in increasing order
2. T <— null
3. for each u in V make a set containing singleton u
4. for i = 1 to m
   1. (u, v) = e(i)
   2. if (u and v are in different sets)
      1. T <— T union {e(i)}
      2. merge the sets containing u and v
5. return T

Runtime: O(ElogE) use **min heap**

Dynamic Programming

Knapsack Problem
Algorithm
1. input: n, w1, …, wN, v1, …., vN
2. for w = 0 to W
   1. M[0,w] = 0
3. for i = 1 to n
   1. for w = 1 to W
      1. if (wi > w)
         1. M[i, w] = M[i-1, w]
      2. else
      3. M[i, w] = max { M[i-1, w], vi + M[i-1, w-wi] }
4. return M[n, W]

Runtime: O(nW) [pseudo-polynomial]

Shortest Path with Negative Edges / Bellman-Ford Algorithm
NOTE:
slower than Dijkstra's Algorithm
deal with **negative** weight cycles in the graph

Algorithm
push-based-shortest-path(G, s, t)
1. foreach node v in V
   1. M[v] <— infinity
   2. successor[v] <— null

2. M[t] = 0
3. for i = 1 to n-1
    1. foreach node w in V
        1. if (M[w] has been updated in previous iteration)
            1. foreach node v such that (v, w) in E
                1. if (M[v] > M[w] + c(v,w))
                    1. M[v] <— M[w] + c(v,w)
                    2. successor[v] <— w
        2. if no M[w] value changed in iteration i, stop

Runtime: O(V*E)

Sequence Alignment / Min Edit Distance
Algorithm
Sequence-Alignment(m, n, x1x2…xm, y1y2…yn, a, b)
1. for i = 0 to m
    1. M[0, i] = ia
2. for j = 0 to n
    1. M[j, 0] = ja
3. for i = 1 to m
    1. for j = 1 to n
        1. M[i, j] = min(b[xi, yi] + M[i-1,j-1], a + M[i-1, j], a+ M[i, j-1])
4. return M[m, n]

Runtime: O(mn)

Reduction

- multiple calls
  - e.g.: Common Element
- pre-processing
  - e.g.: Max-heap using Min-heap
- equivalence
  - e.g.: Ship-Port to Stable Matching

P/NP

Basic genres
- packing problems
- covering problems
- constraint satisfaction problems
- sequencing problems
- partitioning problems
- numerical problems

P = problems for which there exists an algorithm with polynomial running time
NP = languages where coming up with a proof might be very hard, but “checking” the answer is always polynomial-time

(Largest) Independent Set
Subset of vertices S in V such that |S| >= k, and for each edge at most one of its endpoints is in S.

(Smallest) Vertex Cover

Subset of vertices S in V such that ISI <= k, and for each edge, at least one of its endpoints is in S.

NOTE:

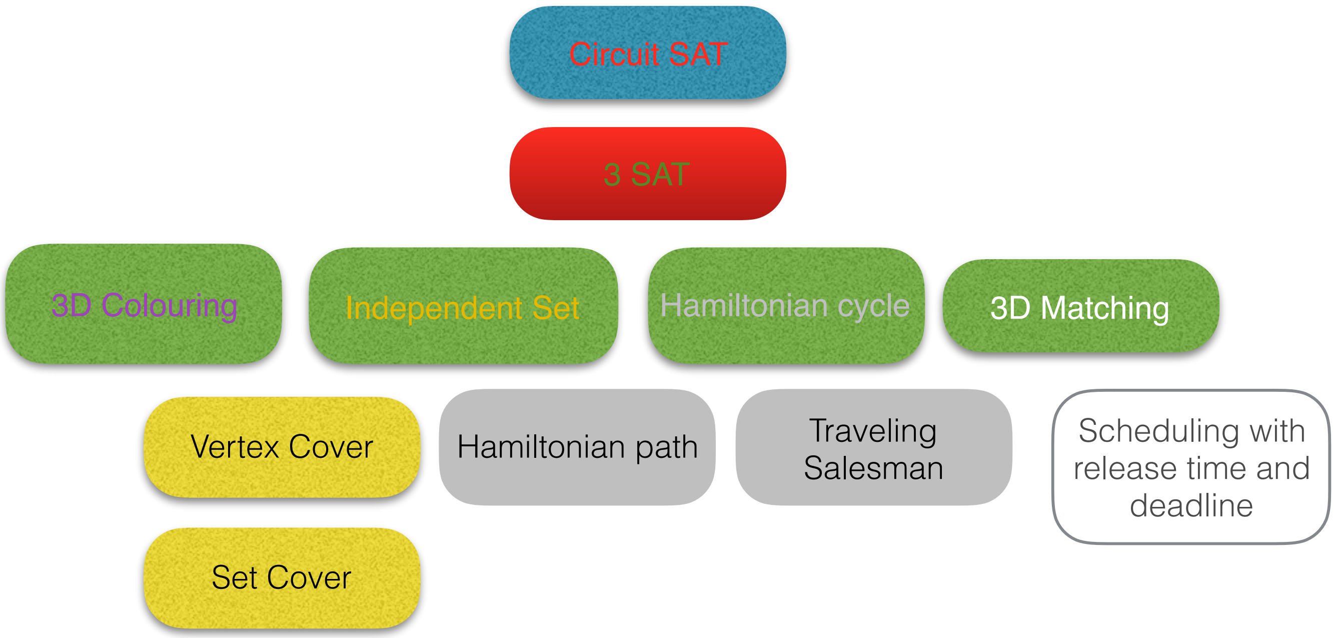
S is an independent set iff V-S is a vertex cover. [equivalence]

Set Cover

Set of U elements. Does there exist a collection of <= k of these sets whose union is equal to U?

NOTE:

Vertex Cover reduces to Set Cover.



Greedy Algorithm

General approach for prove correctness:

1. Prove that there exists an optimum which made the same “first choice” our greedy algorithm made
2. Prove inductively that this means our greedy algorithm is optimum

Interval Scheduling Problem

Algorithm

Interval-Schedule(S)

1. Let A <— the empty set
2. while S is non-empty:
    1. Let i have the earliest finish time in S
    2. Let A <—A union {i}
    3. Let S <— S less all intervals which overlap i
3. Return A

***Greedy: choose the one that ends earliest

Proof Correctness by Induction:

Case 0: True

Case 1: True

Assume True for some n.
Consider case n+1:
We will have OPT(n) #|OPT| intervals that ends the earliest. With a new interval i.
If i does not overlap with OPT(n) set, add i to OPT.
OPT # of interval is |OPT| + 1.
If i overlap with OPT(n) set, because our OPT is choose by earliest end time, so if there exist any other OPT, they also will not be able to include i.
OPT # of interval is |OPT|.
In both cases, we get maximum.

Interval Partitioning Problem

Algorithm

```
Sort intervals by starting time so that s1 < s2 < … < sn.
d <— 0 //number of allocated classrooms
for j = 1 to n {
        if (lecture j is compatible with some classroom k)
                schedule lecture j in classroom k
        else
                allocate a new classroom d + 1
                schedule lecture j in classroom d + 1
                d <— d + 1
}
```

Proof Correctness by Contradiction:

Assume there exist an OPT1 with less than d classrooms.
Because our OPT are chosen based on the number of lectures with the same time, and there are d of them.
If we use less than d classrooms, there will be one lecture during that period of time with no classroom to hold.

Minimising Lateness

Algorithm

```
Sort n jobs by deadline so that d1 < d2 < … < dn
t <— 0
for j = 1 to n
        Assign job i to interval [t, t+ tj]
        sj <— t, fj <— t + tj
        t <— t + tj
output intervals [sj, fj]
```

***NO IDLE TIME
***NO INVERSION

Proof Correctness by Inversion / Exchange argument:

Define S* to be an OPT that has fewest number of inversions.
Can assume S* has no idle time.
If S* has no inversions, then S = S*
If S* has an inversion, let i-j be an adjacent inversion.
Swapping i and j does not increase the maximum lateness and strictly decreases the number of inversions
This contradicts definition of S*

Coin-Changing Problem

Algorithm

Sort coins denominations by value: c1 < c2 < … < cn.
S <— null set // coins selected
while (x != 0) {
    let k be the largest integer such that ck <= x
    if (k == 0)
        return "no solution found"
    x <— x - ck
    S <— S union {k}
}
return S

NOTE:
O(n^3) to determine if coins are Canonical (i.e. greedy works)

Divide and Conquer

Merge-Sort

Algorithm

Merge-Sort(A)

1. If length(A) = 1 then return A
2. Else let B be the first half of A, C be the second half
3. Merge-Sort(B)
4. Merge-Sort(C)
5. A <— Merge(B, C)

Runtime:
T(n) = T(n/2) * 2 + merging
= O(n logn)

Master Theorem

******** TO BE FILLED IN **********

QuickSort

Pick the median and divide the array into bigger and smaller half, recursively.
*** Faster than merge sort if pick the correct median.

Counting Inversions

**Measure the "sortedness" of an array

Algorithm

Sort-and-Count(L) {
    if list L has one element
        return 0 and the list L
    Divide the list into two halves A and B
    (rA, A) <— Sort-and-Count(A)
    (rB, B) <— Sort-and-Count(B)
    (r, L) <— Merge-and-Count(A, B)
    return r = rA + rB + r and the sorted list L
}

Closest-Pair

Algorithm

Closest-Pair(p1, p2, …, pn) {

```
	compute separation line L such that half the points are on one side and half on the other
side

	d1 = Closest-Pair(left half)
	d2 = Closest-Pair(right half)
	d = min (d1, d2)

	delete all point further than d from separation line L

	sort remaining points by y-coordinates

	scan points in y-order and compare distance between each point and next 11 neighbours. If
	any of these distances is less than d, update d.

	return d
}
```

Runtime: O(n (logn))

Maximum Flow & Minimum Cut Problem

Ford-Fulkerson Algorithm

```
Augment(f, c, P) {
	b <— bottleneck(P)
	for each e in P {
		if (e in E) f(e) <— f(e) + b	//forward edge
		else f(eR) <— f(e) - b		//reverse edge
	}
	return f
}

Ford-Fulkerson(G, s, t, c) {
	for each e in E
		f(e) <— 0
	Gf <— residual graph
	while (there exists augmenting path P) {
		f <— Augment (f, c, P)
		update Gf
	}
	return f
}
```

Runtime: O(mn)

***IMPORTANT: Choose GOOD augmenting path!! — **Capacity Scaling**
Gf be the subgraph consisting of only arcs with capacity at least the scaling parameter c.
1 < c < depends
when c == 1 —> max flow
Runtime: (m^2 logC)

Max-Flow Min-Cut Theorem
Value of Max Flow == Value of Min Cut.
Flow f is a max flow iff there are no augmenting paths.
The following 3 are equivalent:

- There exist a cut (A, B) such that v(f) = cap(A, B)
- Flow f is a max flow
- There is no augmenting path relative to f

Preflow-Push Algorithm for Maximum Flow

NO SHARP DROPS

Algorithm

Start with labelling: h(s) = n, h(t) = 0, h(v) = 0
Start with freflow: f(e) = c(e) for e = (s, v), f(e) = 0 otherwise
While there is a node (other than t) with positive excess
        Pick a node v with excess(v) > 0
        If there is an edge (v, w) in Ef such that push(v, w) applies
                Push(v, w)
        Else
                Relabel(v)

Push(v, w):
        Applies if excess(v) > 0, h(w) < h(v), q = min(excess(v), cf(v, w))
        Add q to f(v, w)

Relabel(v):
        Applies if excess(v) > 0 and for all w such that in Ef, h(w) >= H(v)
        Increase h(v) by 1

Bipartite Matching

Edge Disjoint Paths

Randomized Algorithms

RP

L belong to RP if there exists probabilistic Alg in such that
- If x is not in L, Alg(x) always outputs NO
- if x is in L, Alg(x) outputs YES with prob > 2/3

Co-RP

L belongs to co-RP if there exists probabilistic Alg such that
- if x is in L, Alg(x) always output YES
- if x is not in L, Alg(x) outputs NO with prob > 2/3

BPP

L belongs to BPP if there exists probabilistic Arg such that
- if x is in L, Alg(x) outputs YES with prob > 2/3
- if x is not in L, Alg(x) outputs NO with prob > 2/3