FLOW NETWORK

A flow network is a diagraph G=(V,E) with a sourch and sing and non negative integer capacity c(e) for each e in E

NO PARALLEL EDGES NO EDGE ENTERS S NO EDGE LEAVES S

A s-t cut is a partition (A,B) of the vertices with s and t in A and B

The capacity of an s-t cut is the sum of the capcities of the edges from A to B

An s-t flow is a function from the edges to the integers satisfying the following:

Every edge e in E has a flow less than or equal to capacity

Every vertex v is not part of s,t

Flow from s is same as flow into t

BiPartite Matching:

Given an undirected graph G-(V,E) of a subset of edges M from E is a matching if each node appears in at most one edge in E

SOLVE:

1. Consider a flow network G' where all edges are directed from L to R and match the edges in G
2. There are edges from s to every vertex in L
3. There are edges to t from every vertex in R
4. C(e) = 1 for every e in E'

FORD FULKERSON ALGORITHM

Input: Directed Graph G=(V,E) with capacity c in E->N+

Source s and sink t

Output is the max flow

ALGO:

PICK A PATH from s to t, flow on it as much as possible

Update network

Create a residual network – if there is a simple path, flow more!

Every iteration takes linear time in the size of the graph.

Maximality of flow: why does getting stuck mean we found max flow?

Flow value lemma: Let f be a flow : Let f be a flow: (C, V-C) be an s-t cut, sin C< t in V-C

Min Cut – Max Flow Thereom

Value of max flow = capacity of Min cut

In fact: The following are equivalent

1. There is an st cut whose capacity is the value of f.
2. F is a max flow
3. 3. There are no augmenting paths in respect to f

EDIT DISTANCE:

Given an alignment of two strings:

It's cost is the sum of the number of gaps and number of mismatches. The edit distance between two strings is the min cost of of an alignment between them.

1. Align x and y; Cost: possible mismatch between x_1 + y_j plus cost of aligning x_1..x_i-1 & y_1 – y_j-1
2. Align x1 – x_i-1 and y_1 – y_j leave x_i unmatched cost: 1 for gap + cost of aligning x_1-x_i-1, y_1-y_j
3. Similarly but when y_i is unmatched

OPT(i,0) = I OPT(0,j) = j

OPT(I,j) = min{ (o if x_i = y_j, 1 if x_i != y_j) + OPT(i-1, j-1), 1+OPT(i-1,j), 1+OPT(I,j-1))

Algorithm:

```
For I = 0..m
	Opt[I,0] = i
For j = 0..n
	OPT(0,j] = j
For I = 1..m
	For j=1..n; b = (x_i = y_j) OPT[I,j] = min{b+OPT[i-1,j-1], 1+OPT[i-1,j], 1+OPT[I, j-1]
```

Knapsack
Input: Items 1..n where each have value v_i and weight w_i greater than 0. Capacity of W.
Output: Fill the knapsack with the max value where you don't exceed the capacity.
OPT(i,w) – max profit for object 1..i and weight <= w
OPT(0,w) = 0
OPT(I,w) = max {
  OPT(i-1, w)
  v_i + OPT(i-1, w-w_i)
Algorithm:
For w = 0..W
    OPT[0,w] = 0
For I = 1..n
    For w = 1..w
    If (w_1 > w)
    OPT[I,w] = OPT[i-1,w]
    ELSE
(ABOVE CONDITIONAL)
Runtime is O(nxW)

Interval Scheduling:
Input: Tasks with weight w_i
Output: Non overlapping tasks with total max weight.
Sort jobs by finish time and OPT[j] has value of optimal solution up to f_j.
Therefore: algo is
OPT[j] = max (w_j + OPT[p(j)], OPT[j-1]) where OPT[o] = 0.
You either choose to do the task or not.
O(nlgn) to sort by finish time.
O(nlgn) to comput p)
Overall: O(nlgn)
Algorithm:

1. Sort f_1..f_n
2. Compute p(1)..p(n)
3. OPT[0] = 0
4. For j=1..n
   Opt[j] = max{w_j + OPT(p(j)), OPT[j-1]

Elements of Dynamic Programming:

- Order/Time
- Polynomilaity

ShortestPaths
Input: Weighted directed graph G=(V,E). there are negative weights but no negative cycles
Output: length of shortest path from s to t
OPT(I,v) = length of shortest v-t path using <=I edges
OPT(I,v) = min { OPT(i-1,v), OPT(i-1,u)+W(v,u) }
OPT(0,v) = infinity OPT(0,t) = 0
NOTE: Length of shortest path <= n-1, otherwise there's a cycle.
Algorithm:
For each v in V
    OPT[0,v] = infinity
OPT[0,t] = 0
For I = 1 .. n-1
    For each v in V
        OPT[I,v] = OPT[i-1,1]
    For each (v,u) in E
        OPT[I,v] = ABOVE ALGO
TIME: O(V X E) + V^2
SPACE: O(VXE)
BELLMAN FORD:
For each v in V
    OPT[v] = infinity
OPT[t] = 0
For I = 1..n-1
    For each (u,v) in E
        If OPT[v] + W(u,v) < OPT[u]
            OPT[u] = OPT[v] + W(u,v)
TIME:O(VXE) SPACE:O(V)
Check if BF has neg cycles by adding this to end
For each (u,v) in E
    If OPT[v] = w(u,v) < OPT[u]
        Then announce negative cycle

Dynamic Programming:
Idea: Store OPT[j] in an array. Update tasks takes O(1) time, having computed OPT[i] for every i<j already. This is memorization.