-- Intro

• **Stable Matching** (perfect matching with No unstable pairs). • **Stability** (no incentive for some pair of pariticpants to undermine assignment by joint action)

• **Propose & Reject Algorithm (Gale-Shapley)**

```
Initialize each person to be free.
while (some man is free and hasn't proposed to every woman) {
   Choose such a man m
   w = 1st woman on m's list to whom m has not yet proposed
   if (w is free)
      assign m and w to be engaged
   else if (w prefers m to her fiancé m')
      assign m and w to be engaged, and m' to be free
   else
      w rejects m
}
```

O(n*2)

• [Proofs; by Contradiction/ just ()]

--• mx |Interval Scheduling|
(nlogn - greedy)

• mx|Weighted Interval Scheduling| (nlogn - dynamic)

• mx | Bipartite Matching |
(n^k max-flow algs)

• mx | Independent Set |
(NP-Complete)

-- **Analysis**

- O(n), upper bound
- Ω(n), lower bound
- Ø(n), tight bounds
- Linear Time - O(n)
- O(n.logn) {from div&con}
- Quadratic T - O(n^2)
- Cubic T - O(n^3)
- Polynomial T - O(n^k) [Enumerate All Subsets of K Nodes] (e.g. are there k nodes such that no 2 are joined by an edge)

$$\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k(k-1)(k-2)\cdots(2)(1)} \le \frac{n^k}{k!}$$

- Exponential T - O(2^n) [Enumerate All Subsets] (e.g. max size of indi. set)

-- **Graphs**

• ***BFS*** - L0 {S} - L1 {neighbors of L0}++
O(n^2) > **O(M+N)** • - when we consider node u, there are deg(u) incident edges (u, v) - total time processing edges is $\sum_{u \in V} deg(u) = 2m$

• An undirected G=(V,E) is ***BiPartite*** if nodes can be colored red/blue such that every Edge has 1 red/blue end. • bipartite graphs cannot cntain an Odd length cycle.

• A graph is **Strongly Connected**, if every node is reachable from s, and s is reachable from every node.

• A **DAG (Directed Acyclic Graph)** contains no directed cycle, has a **topological order/s.**
FIND Topological Order of DAG: O(m+n)
BY: maintaining list of nodes with no i / o.

Greedy Algorithms

• Interval Partitioning:

Greedy algorithm. Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

```
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
}
```

• Paths:

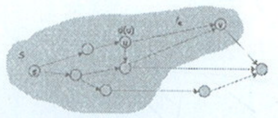
Dijkstra's algorithm. O(mlogn)

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e=(u,v):u\in S} d(u) + \ell_e,$$

add v to S, and set d(v) = π(v).

shortest path to some u in explored part, followed by a single edge (u, v)



• MST: can hv -ve edge weights

Kruskal's algorithm. Start with T = ϕ. Consider edges in ascending order of cost. Insert edge e in T unless doing so would create a cycle.

Reverse-Delete algorithm. Start with T = E. Consider edges in descending order of cost. Delete edge e from T unless doing so would disconnect T.

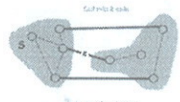
Prim's algorithm. Start with some root node s and greedily grow a tree T from s outward. At each step, add the cheapest edge e to T that has exactly one endpoint in T. - works for undirected.

O(n²) w/ Array, O(mlogn) w/ Binary Heap

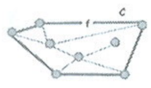
Simplifying Assumption: All edge Costs (Ce) are distinct.

Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST contains e.

Cycle property. Let C be any cycle, and let f be the max cost edge belonging to C. Then the MST does not contain f.



e is in the MST



f is not in the MST

Claim: A cycle & a cutset intersect in an EVEN number of edges.
Pfs. Cut/Cycle Properties; Exchange Argument (Contradiction)

Implementation: Kruskal's Algorithm

Implementation. Use the union-find data structure. (new edges added w/in independent components)

- Build set T of edges in the MST.
- Maintain set for each connected component.
- O(m log n) for sorting and O(m α (m, n)) for union-find. usually DFS/BFS = Cycle Tests

m ≤ n² → log m is O(log n)   essentially a constant

M≤n² → logm is O(logn)

```
Kruskal(G, c) {
   Sort edges weights so that c1 ≤ c2 ≤ ... ≤ cm.
   T ← ϕ

   foreach (u ∈ V) make a set containing singleton u

   for i = 1 to m          are u and v in different connected components?
      (u,v) = ei
      if (u and v are in different sets) {
         T ← T ∪ {ei}
         merge the sets containing u and v
      }                    merge two components
   return T
}
```

To remove the assumption that all edge costs are distinct: perturb all edge costs by tiny amounts to break any ties.

^ (for ~~Prim's~~ / Kruskal's.)

MST Algorithms: Theory: Deterministic Comparison Based Algs:

- O(m log n)   [Jarník, Prim, Dijkstra, Kruskal, Boruvka]

Divide & Conquer

{ Brute=n^2 , D&C=nlogn }

• Merge Sort: ÷Array into 2, Recursively sort each half, merge two halves. • O(n.logn)

• Counting Inversions: O(nlogn)

• Closest Pair Algorithm: O(nlogn)

Dynamic Programming ≠ recursively implemented

• Weighted Interval Scheduling: • [Greedy Algorithm can fail (alot) if arbitary weights are allowed] • [Brute Force, Recursion, Fails due to Redundant Sub-Probs - Exponential Algs.] • Soln: Memorzation

• Binary Choice:

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{\underbrace{v_j + OPT(\underbrace{p(j)}_{\text{next available}})}_{\text{Take}},\ \underbrace{OPT(j-1)}_{\text{Leave}}\} & \text{otherwise} \end{cases}$$

Weighted Interval Scheduling: Running Time

Claim. Memoized version of algorithm takes O(n log n) time.

- Sort by finish time: O(n log n).
- Computing p(·): O(n) after sorting by start time.

- M-Compute-Opt(j): each invocation takes O(1) time and either
  - (i) returns an existing value M[j]
  - (ii) fills in one new entry M[j] and makes two recursive calls

- Progress measure Φ = # nonempty entries of M[].
  - initially Φ = 0, throughout Φ ≤ n.
  - (ii) increases Φ by 1 ⇒ at most 2n recursive calls.

- Overall running time of M-Compute-Opt(n) is O(n). ▪

Remark. O(n) if jobs are pre-sorted by start and finish times.

↑(unwind recursion)

Input: $n, s_1 \ldots s_n, f_1 \ldots f_n, v_1 \ldots v_n$

Sort jobs by f_i; $f_1 \le f_2 \ldots \le f_n$

Compute p(1), p(2) ... p(n)

```
Iterative-Compute-Opt {
   M[0] = 0
   for j=1 to n
      M[j] = max(vj + M[p(j)], M[j-1])
}
```

-- Segmented Least Squares:

- Find a line y = ax + b that minimizes the sum of the squared error:

$$SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min_{1 \le i \le j} \{ e(i,j) + c + OPT(i-1) \} & \text{otherwise} \end{cases}$$

Segmented Least Squares: Algorithm

```
INPUT: n, p1,...,pN , c

Segmented-Least-Squares() {
   M[0] = 0
   for j = 1 to n
      for i = 1 to j
         compute the least square error eij for
         the segment pi,..., pj

   for j = 1 to n
      M[j] = min 1 ≤ i ≤ j (eij + c + M[i-1])

   return M[n]
}
```

Running time. O(n³). ← can be improved to O(n²) by pre-computing various statistics

- Bottleneck = computing e(i, j) for O(n²) pairs, O(n) per pair using previous formula.

• Knapsack

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{OPT(i-1,w),\ v_i + OPT(i-1, w-w_i)\} & \text{otherwise} \end{cases}$$

Knapsack Problem: Bottom-Up ↑

Knapsack. Fill up an n-by-W array.

```
Input: n, w1,...,wN, v1,...,vN

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}

return M[n, W]
```

Running time. Θ(n W).

- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete.

|W| = log (W)
$2^{|W|} = 2^{\log(W)} = W$
∴ input is exponential (bad).

• Shortest Paths:

$$OPT(i,v) = \begin{cases} 0 & \text{if } i = 0 \\ \min\left\{OPT(i-1,v),\ \min_{(v,w)\in E}\{OPT(i-1,w) + c_{vw}\}\right\} & \text{otherwise} \end{cases}$$

Shortest Paths: Implementation

```
Shortest-Path(G, t) {
   foreach node v ∈ V
      M[0, v] ← ∞
   M[0, t] ← 0

   for i = 1 to n-1
      foreach node v ∈ V
         M[i, v] ← M[i-1, v]
      foreach edge (v, w) ∈ E
         M[i, v] ← min { M[i, v], M[i-1, w] + cvw }
}
```

Analysis. Θ(mn) time, Θ(n²) space.

Finding the shortest paths. Maintain a "successor" for each table entry.

- Improve: maintain only 1 array M[v] = cur shortest path

Overall impact.

- Memory: O(m + n).
- Running time: O(mn) worst case, but substantially faster in practice.

Bellman-Ford: Efficient Implementation
(-ve paths: ✓)
(djkstra(x))

```
Push-Based-Shortest-Path(G, s, t) {
   foreach node v ∈ V {
      M[v] ← ∞
      successor[v] ← ϕ
   }

   M[t] = 0
   for i = 1 to n-1 {
      foreach node w ∈ V {
         if (M[w] has been updated in previous iteration) {
            foreach node v such that (v, w) ∈ E {
               if (M[v] > M[w] + cvw) {
                  M[v] ← M[w] + cvw
                  successor[v] ← w
               }
            }
         }
         If no M[w] value changed in iteration i, stop.
   }
}
```

Detecting Negative Cycles: Summary

Bellman-Ford. O(mn) time, O(m + n) space.

- Run Bellman-Ford for n iterations (instead of n-1).
- Upon termination, Bellman-Ford successor variables trace a negative cycle if one exists.

(n-1) passes

Greedy Algorithms

[Interval Scheduling]

Greedy Template: Consider jobs in some order. Take each job provided it's compatible w/ already taken.

(Earliest Start Time) $\uparrow_{asc} s_j$

(Earliest Finish Time) $\uparrow f_j$

(Shortest Interval) $\uparrow (f_j - s_j)$

(Fewest Conflicts) $\uparrow c_j$

[Minimizing Lateness]

Greedy Template: Consider jobs in some order

(Shortest processing time first) $\uparrow t_j$

(Earliest deadline first) $\uparrow d_j$

(Smallest Slack) $\uparrow (d_j - t_j)$

Maximum Matching
contains the largest possible number of edges

[illegible]

Master Theorem (for Recurrence Relations)

If $T(n) \leq a \cdot T(\frac{n}{b}) + O(n^d)$

then

$$T(n) = \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log n) & a = b^d \\ O(n^{\log_b a}) & a > b^d \end{cases}$$

// for Merge Sort

$1 \leq k [\leq 2^k T(\frac{n}{2^k}) + k\, cn]$

let $k = \log n$

$a = 2, b = 2, d = 1$

via MT: $O(n \log n)$

// for Matrix Multiplication

$M(n) = 8M(\frac{n}{2}) + O(n^2)$

$a = 8, b = 2, d = 2$

via MT:

$O(n^{\log_2 8}) = O(n^3)$

Max Flow = Min Cut

[illegible]

Prim's Algorithm

Prim(G, c) {

for each ($v \in V$) ...

...priority queue Q

for each ($v \in V$) insert v into Q

Initialize set of explored nodes $S \leftarrow \emptyset$

while (Q !empty) {

$u \leftarrow$ delete min element from Q

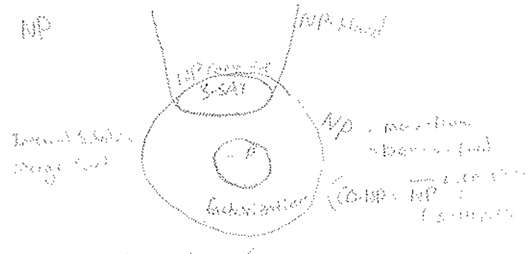
$S \leftarrow S \cup \{u\}$

for each (edge $e = (u, v)$) incident to u

if ($v \notin S$) && ($c_e < a[v]$)

...

}



X is NP complete if

(a) $X \in NP$

(b) for all $A \in NP$, $A \leq_p X$

if only (b), X is NP-Hard

NP: Instance ...

3SAT

reduces to

(max) Independent Set

(min) Vertex Cover

Set Cover

[illegible]