BACKGROUND

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

- O(n), upper bound • Ω(n), lower bound • Ø(n), tight bounds
- O(n.logn) {from div&con) • 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)
- Exponential T - O(2^n) [Enumerate All Subsets] (e.g. max size of indi. set)

• ***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 $\Sigma_{u \in V} deg(u) = 2m$

• 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 cant cntain 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: maintain list of nodes with no i / o.

• Arrays are Invertable

• Maximum Independent Set = !Minimum Vertex Cover
(Minimum Vertexes required to cover all EDGES)

GREEDY

+(Can be Optimal in Some cases)

+Interval Scheduling:
Earliest start, Earliest finish, shortest interval, fewest conflicts

+Interval Partitioning:
(Lower Bound, Depth of set is max num contained any given time)
rooms in ^startTime, insert to room if free, otherwise create room

+Minimizing Lateness:
Shortest Procesing Time First, Earliest Deadln First, Smllst Slack

Dijkstra's algorithm.

- 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)

_Array, Binary Heap, d-way Heap, Fib Heap :: n^2, mlogn, $m\log_{(m/n)}n$, m + nlogn
+MST:

Kruskal's algorithm. Start with T = ϕ (Nothing). Consider edges in ascending order of cost. Insert edge e in T unless doing so would create a cycle.
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.

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.

+ Claim: A cycle & a cutset intersect in an EVEN number of edges.
Pfs. Cut/Cycle Properties; Exchange Argument (Contradiction)
+ Lexicographic Tiebreaking (for K&P): To remove the assumption that all edge costs are distinct: peturb all edge costs by tiny amounts to break any ties.

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

DIVIDE & CONQUER

[+ MergeSort, Counting Inversions, Closest-Pair, [All O(nlogn)]
+Sequence Alignment-
+MergeSort:: Divide O(1) ++ Sort 2T(n/2) ++ Merge O(n)
+Counting Inversions: v-A B C-v, >B (A) C>, [A}B,...]
+Strassen's Matx Mx: ([C11,C12,C21,C22]=[A11...]x[B11...]) (C11 = (A11xB11)+(A12xB21)..)} (C11=P5+P4-P2+P6)
+Master Theorem:

if $T(n) \le aT(n/b) + O(n^d)$

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

T(n) = a.T(n/b) + O(n^d), where a≥1, b>1.
• n is the size of the problem.
• a is the number of subproblems in the recursion
• n/b is the size of each subproblem (assumed same size)
• n^d is the work done outside the recursive calls (+ dividing+merging)

DYNAMIC PROGRAMMING

>> Weighted Interval Scheduling [O(nlogn)]

"Greedy Algorithm fails with arbitary weights, Brute Force uses redundant sub-probs; ie Memorize"

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

>> Segmented Least Squares

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

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

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

[O(n^3), improvable to O(n^2) by pre-computing]

>> Knapsack

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

w = limitCurW
wi = itemCurW

{ v-{i,t,e,m,s}-v , >-(weight)->, [[v][a][l][u][e]] :: OPT{p,k,d}, value=.. }
n+logW input. Ø(nW). Decision version of Knapsack is *NP-Complete.*
|W| = log(W), 2^|W| = 2^logW = W, input=exponential :[

>> Sequence Alignment

o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

1 mismatch, 1 gap

$$OPT(i,j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

1. OPT matches xi-yj, 2. leaves xi unmatched, 3. leaves yj unmatched

>> Shortest Path

[O(n^2) space, O(mn) time]

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

>> Bellman-Ford

"Can detect –ve cycles. Run for n iterations (instead of n-1), on termination, successor vars trace a –ve cycle if 1 exists"
[O(m+n) space, O(mn) time]

Bellman-Ford: Efficient Implementation

```
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 {for each node leading to (w)
            if (M[v] > M[w] + cvw) {     C(vw) = cost
               M[v] ← M[w] + cvw
               successor[v] ← w
            }
         }
      }
      If no M[w] value changed in iteration i, stop.
   }
}
```

FLOW

Def. The capacity of a cut (A, B) is: $cap(A, B) = \sum_{e \text{ out of } A} c(e)$

Def. An s-t flow is a function that satisfies:

- For each e ∈ E: $0 \le f(e) \le c(e)$ (capacity) (limits)
- For each v ∈ V – {s, t}: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ (conservation)

Def. The value of a flow f is: $v(f) = \sum_{e \text{ out of } s} f(e)$.

Flow value lemma. Let f be any flow, and let (A, B) be any s-t cut. Then, the net flow sent across the cut is equal to the amount leaving s.

$$\sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) = v(f)$$

Weak duality. Let f be any flow, and let (A, B) be any s-t cut. Then the value of the flow is at most the capacity of the cut.

Corollary. Let f be any flow, and let (A, B) be any cut.
If v(f) = cap(A, B), then f is a max flow and (A, B) is a min cut.

>> Ford Fulkerson (Augmenting Path Algorithm)

```
Augment(f, c, P) {
   b ← bottleneck(P)
   foreach e ∈ P {
      if (e ∈ E) f(e) ← f(e) + b      forward edge
      else       f(eR) ← f(e) - b     reverse edge
   }
   return f
}
```

```
Ford-Fulkerson(G, s, t, c) {
   foreach e ∈ E  f(e) ← 0
   Gf ← residual graph

   while (there exists augmenting path P) {
      f ← Augment(f, c, P)
      update Gf
   }
   return f
}
```

Augmenting path theorem. Flow f is a max flow iff there are no augmenting paths.

Max-flow min-cut theorem. [Ford-Fulkerson 1956] The value of the max flow is equal to the value of the min cut.

(min-cut has to be through edges that are all 0(full))

Corollary. If C = 1 (Capacity), Ford-Fulkerson runs in O(mn) time.

[O(m.f), f = maxFlow; each augmenting path found in at most O(m) time, increasing flow by at least 1]
[O(nm^2), via E. Karp; define search order, scale]

Capacity Scaling

Intuition. Choosing path with highest bottleneck capacity increases flow by max possible amount.

- Don't worry about finding exact highest bottleneck path.
- Maintain scaling parameter Δ.
- Let $G_f(\Delta)$ be the subgraph of the residual graph consisting of only arcs with capacity at least Δ.

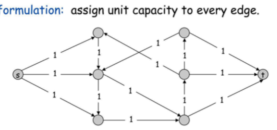
```
Scaling-Max-Flow(G, s, t, c) {
   foreach e ∈ E  f(e) ← 0
   Δ ← smallest power of 2 greater than or equal to C
   Gf ← residual graph

   while (Δ ≥ 1) {
      Gf(Δ) ← Δ-residual graph
      while (there exists augmenting path P in Gf(Δ)) {
         f ← augment(f, c, P)
         update Gf(Δ)
      }
      Δ ← Δ / 2
   }
   return f
}
```

>> Edge Disjoint Paths

"Given a digraph, with s,t, find max number of edge-disjoint (unique edges) s-t paths"

Max flow formulation: assign unit capacity to every edge.

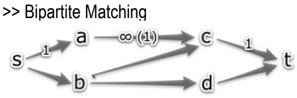


Theorem. Max number edge-disjoint s-t paths equals max flow value.

>> Disconnecting a Network

Theorem. [Menger 1927] The max number of edge-disjoint s-t paths is equal to the min number of edges whose removal disconnects t from s.

>> Bipartite Matching



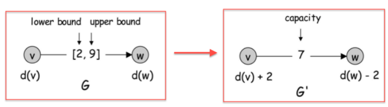
Max |Matching| = Max Flow
(Maximum Matching. Contains the largest possible number of edges)

>> Circulation with Demands, Lower Bounds

Max flow formulation.

- Add new source s and sink t.
- For each v with d(v) < 0, add edge (s, v) with capacity -d(v).
- For each v with d(v) > 0, add edge (v, t) with capacity d(v).
- Claim: G has circulation iff G' has max flow of value D.

(D. saturates all edges leaving s and entering t)



>> Survey Design

Survey design.

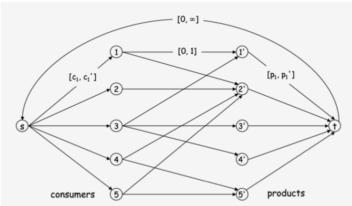
- Design survey asking n_1 consumers about n_2 products.
- Can only survey consumer i about a product j if they own it.
- Ask consumer i between c_i and c_i' questions.
- Ask between p_j and p_j' consumers about product j.

Goal. Design a survey that meets these specs, if possible.

Bipartite perfect matching. Special case when $c_i = c_i' = p_i = p_i' = 1$.

Algorithm. Formulate as a circulation problem with lower bounds.

- Include an edge (i, j) if customer own product i.
- Integer circulation ⇔ feasible survey design.



>> Projection Design

Min cut formulation.

- Assign capacity ∞ to all prerequisite edge.
- Add edge (s, v) with capacity p_v if $p_v > 0$.
- Add edge (v, t) with capacity $-p_v$ if $p_v < 0$.
- For notational convenience, define $p_s = p_t = 0$.

COMPLEXITY

"A problem is NP iff there exists a verifier for the problem that executes in polynomial time."
"For a problem P, we can ignore the certificate, and just solve in polynomial time"
("A proof certificate can simply be a list, can return True or False.")

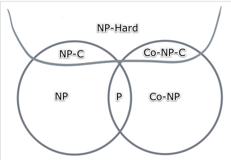
Ex. OR AND

$(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee x_2 \vee x_4) \wedge (\overline{x_1} \vee \overline{x_3} \vee \overline{x_4})$

3 SAT — instance s

$x_1 = 1,\ x_2 = 1,\ x_3 = 0,\ x_4 = 1$

certificate t



"P. Decision Problems for which there is a poly-time algorithm."
"NP. Decision Problems for which there is a poly-time certifier."
"EXP. Decision problems for which there is an exponential-time algorithm. (((P)NP)EXP)"
"IF P=NP: ((P=NP) EXP), if True; Efficient algorithms for 3-Color, TSP, Sat, Factor (breaking RSA cryptography and potentially collapsing economy), but probably not."
"CO-NP is NOT the complement of NP, it IS the SET of the complements of All problems in NP"
"3Sat is the satisfiability problem for CNF (conjunctive normal form) boolean formulas where all clauses have exactly 3 literals."
"A ≡p B means that A and B are polynomially equivalent."
"NP-Complete. A problem in NP such that every problem in NP polynomial reduces to it."
"NP-Hard. A decision problem such that every problem in NP reduces to it."

>> Proof NP-Completeness

1. Show :Prob.: is NP
   (Describe 'Yes-Certificate' and verifiable in P time)
2. Reduce known NP-Complete Problem to (≤p) :Prob.:
3. Show reduction is a Polynomial function.

>> Independent Set (≥p 3-Sat)

3 Satisfiability Reduces to Independent Set

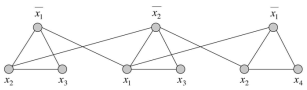
<G> <G , K>

Claim. 3-SAT $\le_p$ INDEPENDENT-SET.
Pf. Given an instance Φ of 3-SAT, we construct an instance (G, k) of INDEPENDENT-SET that has an independent set of size k iff Φ is satisfiable.

Construction.

- G contains 3 vertices for each clause, one for each literal.
- Connect 3 literals in a clause in a triangle.
- Connect literal to each of its negations.

G



k = 3 $\quad \Phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$

>> Weighted Independent Set (≥p Independent Set)

"Reduces from Independent Set with Weights set to 1"

>> Vertex Cover (≥p Independent Set)

Claim. VERTEX-COVER $\equiv_p$ INDEPENDENT-SET.
Pf. We show S is an independent set iff V – S is a vertex cover.

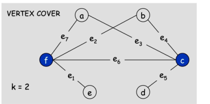
>> Set Cover (≥p Vertex Cover)

Vertex Cover Reduces to Set Cover

Claim. VERTEX-COVER $\le_p$ SET-COVER.
Pf. Given a VERTEX-COVER instance G = (V, E), k, we construct a set cover instance whose size equals the size of the vertex cover instance.

Construction.

- Create SET-COVER instance:
  - k = k, U = E, $S_v = \{e \in E : e \text{ incident to } v\}$
- Set-cover of size ≤ k iff vertex cover of size ≤ k. ▪



SET COVER

U = { 1, 2, 3, 4, 5, 6, 7 }
k = 2
$S_a = \{3, 7\}$ $\quad S_b = \{2, 4\}$
$S_c = \{3, 4, 5, 6\}$ $\quad S_d = \{5\}$
$S_e = \{1\}$ $\quad S_f = \{1, 2, 6, 7\}$

>> Directed-Hamiltonian-Cycle (≥p 3-Sat)

"Hamiltonian-Cycle. given an undirected graph, does there exist a simple cycle that contains every node V."

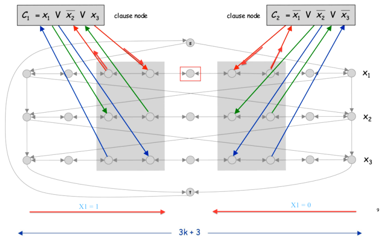
3-SAT Reduces to Directed Hamiltonian Cycle

Construction. Given 3-SAT instance Φ with n variables x_i and k clauses.

- Construct G to have 2^n Hamiltonian cycles.
- Intuition: traverse path i from left to right ⇔ set variable $x_i = 1$.

Construction. Given 3-SAT instance Φ with n variables x_i and k clauses.

- For each clause: add a node and 6 edges.



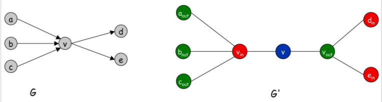
>> (Undirected) Hamiltonian-Cycle (≥p D. Ham-C)

Directed Hamiltonian Cycle

DIR-HAM-CYCLE: given a digraph G = (V, E), does there exists a simple directed cycle Γ that contains every node in V?

Claim. DIR-HAM-CYCLE $\le_p$ HAM-CYCLE.

Pf. Given a directed graph G = (V, E), construct an undirected graph G' with 3n nodes.



>> Traveling Sales Person (≥p Hamilton-Cycle) (metric)

Traveling Salesperson Problem

TSP. Given a set of n cities and a pairwise distance function d(u, v), is there a tour of length ≤ D?

HAM-CYCLE: given a graph G = (V, E), does there exists a simple cycle that contains every node in V?

Claim. HAM-CYCLE $\le_p$ TSP.
Pf.

- Given instance G = (V, E) of HAM-CYCLE, create n cities with distance function
$$d(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E \\ 2 & \text{if } (u, v) \notin E \end{cases}$$
- TSP instance has tour of length ≤ n iff G is Hamiltonian. ▪

Remark. TSP instance in reduction satisfies Δ-inequality.

>> Longest Path (≥p Hamiltonion-Cycle)

"Claim. Hamiltonian Path ≤ p Longest Path (This construction of Hamiltonian Path leads to is a special case of Longest Path)
We have a graph G that contains a Hamiltonian Path, if and only if G has a longest path of length |V |- 1.

G contains a Hamiltonian Path ⇒ G has a Longest Path of length |V |- 1.

Proof. Assume G is not a Hamiltonian path of size |V |, then it means G visits all vertices, which means there exists a path of which length is |V |- 1. This is exactly the definition of LP. __

G has a Longest Path of length |V |- 1 ⇒ G contains a Hamiltonian Path.

Proof. Conversely, if G forms a Longest Path of size |V |- 1, then we know that a simple path of length n- 1 must contain n vertices and hence must be a Hamiltonian Path. __"

>> Clique (≥p 3-Sat)

"A Complete Graph is called a Clique"

In order to prove that CLIQUE is NP-complete we need to prove that CLIQUE is in NP and that CLIQUE is NP-hard (a known NP-complete problem reduces to it)

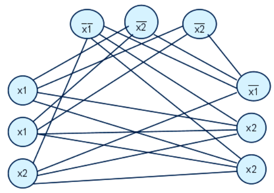
First of all, CLIQUE is in NP. A yes-certificate for an instance <G,k> where G=(V,E) is simply a subset of vertices C ⊆ V of size k that is a clique. In order to check that the certificate is correct in polynomial time, verify that there are k vertices in it, and also verify that for every pair of vertices u,v ∈ C, there is an edge between them (u,v)∈ E.

Therefore CLIQUE ∈ NP. It remains to show that CLIQUE is NP-hard.
We reduce from 3SAT to show that 3SAT le;p CLIQUE.

We use the following reduction. Given an instance φ of 3SAT we build a graph G as follows: for each clause in the formula, which has three literals, introduce a new vertex for each literal labeled by the literal it stands for. If there are k clauses in phi; then there will be k groups of three vertices. Now add edges that connect only compatible literals and only across clauses (groups of 3 vertices). a literal is compatible with any other literal except with its negation. So basically we connect every literal with every other literal in other clauses except its negations. Here is an example:

3SAT instance: φ = (x_1 ∨ x_1 ∨ x_2) ∧ (x_1 ∨ x_2 ∨ x_2) ∧ (x_1 ∨ x_2 ∨ x_2)

transforms to the following instance of CLIQUE, with k=3 and G as follows



So given an instance phi; of 3SAT we construct an instance <G,k> of CLIQUE where G is constructed as above and k = the number of clauses in φ.

Now we need to argue that this transformation works both directions:
(1) if φ has a satisfying assignment then <G,k> has a clique of size k and
(2) if <G,k> has a clique of size k then φ has a satisfying assignment

The correspondence between truth assignments and cliques is almost immediate:
(1) suppose φ has a satisfying assignment. In that assignment at least one literal in each clause is true. that's k literals that can be set to true at the same time and make φ true. those literals correspond to vertices in G, one vertex in each triple. the corresponding variables are compatible (since they are included in a valid assignment) and therefore there are edges between all of them. that makes a clique of size k in G.

(2) If G has a clique of size k, that must include exactly one literal in each triple (since vertices in the same triple do not have edges between them and therefore cannot be in a clique). Since there are edges among all those vertices the corresponding variable in φ must be compatible. set them all to true (and the remaining unassigned variables to anything you want). that will give you one true literal in each clause and therefore it is a satisfying truth assignment for φ

INTRACTABILITY

>> Small Vertex Cover
[Brute Force: O(k.n^(k+1))]

Claim. The following algorithm determines if G has a vertex cover of size ≤ k in $O(2^k kn)$ time.

```
boolean Vertex-Cover(G, k) {
   if (G contains no edges)   return true
   if (G contains ≥ kn edges) return false

   let (u, v) be any edge of G
   a = Vertex-Cover(G - {u}, k-1)
   b = Vertex-Cover(G - {v}, k-1)
   return a or b
}
```

Pf.

- Correctness follows previous two claims.
- There are ≤ 2^{k+1} nodes in the recursion tree; each invocation takes O(kn) time. ▪

Recursive Formula

$$T(n, k) \le \begin{cases} cn & \text{if } k = 1 \\ 2T(n, k-1) + ckn & \text{if } k > 1 \end{cases} \Rightarrow T(n, k) \le 2^k c k n$$

Bounded by O(^^^^^) $\quad$ O(2^k.kn)

>> Independent Set on Trees (Maximum)
[O(n), by considering nodes in post-order]

Greedy

```
Independent-Set-In-A-Forest(F) {
   S ← φ
   while (F has at least one edge) {
      Let e = (u, v) be an edge such that v is a leaf
      Add v to S
      Delete from F nodes u and v, and all edges
         incident to them.
   }
   return S
}
```

basically traversing a tree

(List and detach a Leaf), (Delete new Leafs), (Repeat)

>> Weighted Independent Set on Trees
[O(n), visit nodes in postorder, examine each E once]

$$OPT_{in}(u) = w_u + \sum_{v \in \text{children}(u)} OPT_{out}(v)$$
containing u

$$OPT_{out}(u) = \sum_{v \in \text{children}(u)} \max\{OPT_{in}(v),\ OPT_{out}(v)\}$$
not containing u

Dynamic:

```
Weighted-Independent-Set-In-A-Tree(T) {
   Root the tree at a node r
   foreach (node u of T in postorder) {
      if (u is a leaf) {
         Min [u] = wu
         Mout[u] = 0
      }
      else {
         Min [u] = Σv∈children(u) Mout[v]  +  wv
         Mout[u] = Σv∈children(u) max(Mout[v], Min[v])
      }
   }
   return max(Min[r], Mout[r])
}
```

ensures a node is visited after all its children

"Indpendent set on trees. This structured special case is TRACTABLE because we can find a node that BREAKS THE COMMUNICATION among the subproblems in different subtrees."