Stable Matching

- **Definition**: A matching is stable if no unmatched man and woman both prefer each other to their current partners
- **Gale-Shapely Algorithm:**

```
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
}
```

  - Lemma: Men propose to women in <u>decreasing</u> order of preference
  - Lemma: Women stay engaged after the first time they got engaged
  - Lemma: Women's partners keep getting better
  - Lemma: Upon termination, each man is engaged to a unique women
  - Lemma: Upon termination, the matching between men and women is stable
  - The algorithm returns male-optimal stable matching

Asymptotic Analysis

- **Upper**: T(n) = O(f(n))
  If there ∃ constants $c > 0$, $n_0 \geq 0$ s.t. ∀ $n \geq n_0$, $T(n) \leq c \cdot f(n)$
- **Lower**: T(n) = Ω(f(n))
  If there ∃ constants $c > 0$, $n_0 \geq 0$ s.t. ∀ $n \geq n_0$, $T(n) \geq c \cdot f(n)$
- **Tight**: T(n) = Θ(f(n))
  If T(n) = O(f(n)) and T(n) = Ω(f(n))

Transitivity

- f = O/Ω/Θ(g) and g = O/Ω/Θ(h) → f = O/Ω/Θ(h)

Additivity

- f = O/Ω/Θ(h) and g = O/Ω/Θ(h) → f + g= O/Ω/Θ(h)

Rule of Thumbs

$n^n \geq n! \geq c^{kn} \geq c^n \geq n^k \geq nlog(n) \geq n \geq \sqrt{n} \geq log(n) \geq log(log(n))$

$O(log(n!) = O(nlog(n))$

Mathematical Properties

- $a^{logb} = b^{loga}$
- $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \cdots + \frac{1}{n} = O(log(n))$
- GP sum = $\frac{a(r^n - 1)}{r - 1}, |r| > 1$ or $\frac{a}{1-r}, |r| < 1$
- AP sum = $\frac{n}{2}(2a + (n-1)d) = \frac{n}{2}(a + l)$

Common Recurrence Relations

- $T(n) = 2T\left(\frac{n}{2}\right) + n^2 = O(n^2)$
- $T(n) = 2^k T(\sqrt{n}) + c = O(log^k(n))$
- $T(n) = 2^k T\left(n^{\frac{1}{m}}\right) + c = O\left(log^{\frac{k}{lg(m)}}(n)\right)$

The Master Theorem

The **Master Method** depends on the following Theorem:

Theorem: Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence:

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

Then $T(n)$ can be bounded asymptotically as follows.

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, and if $a \cdot f\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

regularity condition

Graphs

	Adjacency List	Adjacency Matrix
Check if (u,v) exists	Θ(deg(u)) = O(V)	Θ(1)
Enumerate all E	Θ(V + E) = O(V^2)	Θ(V^2)
Use cases	Sparse graphs	Dense graphs

Definitions and theorems

- **Trees**:
  - Def: An indirected graph is a tree if it is connected and acyclic
  - Theorem: In any undirected graph G with V nodes, any two statements below imply the third:
    1. G is connected
    2. G is acyclic
    3. G has V – 1 edges

- **Simple path**: A path where all its vertices are distinct
- **Connected Graph (Undirected)**: There is a path between any vertices *u* and *v*
- **Strongly Connected Graph (Directed):** There is a path from *u* to *v* and from *v* to *u* for any pair of vertices *u* and *v*

Graph Traversals

- **BFS**:
  - Use cases
    1. Finding shortest paths
    2. Connectivity
    3. Testing bipartiteness
       A graph is not bipartite if it contains an odd-lengthed cycle
       Use extra 'Color' array and assign color whenever a node is added to L(i+1). After BFS, check for edges for which both ends have the same color
    4. Testing strong connectivity in directed graphs
       BFS G from S. Then, BFS G^{rev} from S. G is strongly connected if both BFS visited all vertices in G
  - Properties:
    1. Edges in graph *G* but not in its BFS Tree *T* all either connect nodes in the same layer in *T* of connect nodes in adjacent layers
- **DFS**:
  - Use cases
    1. Finding the set of all connected components
  - Properties:
    1. For a given recursive DFS(*u*) call, all vertices marked "explored" between the invocation and end of this call are descendants of *u* in the DFS tree *T*
    2. If a graph *G* contains an edge (*u*, *v*) that is not in its DFS tree *T*, then one of *u* or *v* is an ancestor of the other (i.e. diff levels)

```
BFS(s):
    visited[s] ← true
    visited[v] ← false for all other v
    L(0) ← list containing only s
    i ← 0 // layer
    T ← ∅
    while L[i] is not empty do
        L(i + 1) ← empty list
        for each node u ∈ L(i)
            Consider each edge (u, v)
            if visited[v] = false then
                set visited[v] = true
                add edge (u, v) to the tree T
                add v to L(i + 1)
            endif
        endfor
        increment i by one
    endwhile
```

```
DFS(s):
    visited[s] ← false for all v
    S(0) ← stack containing only s
    parent[] ← empty list
    T ← ∅
    while S is not empty do
        pop u from S
        if visted[u] = false then
            set visited[u] = true
            add (u, parent[u]) to T
            for each edge (u, v)
                push v to S
                set parent[v] to u
            endfor
        endif
    endwhile
```

DAGs

- **Properties**:
  - In every DAG, there is a node *v* with no incoming edges
  - A graph G is a DAG if and only if it has a topological ordering
- **Finding a topological ordering**:
  - Kahn's Algorithm [O(V + E)]
    1. Initialise set S that contains all nodes with no incoming edges
    2. Initialise set W to count number of incoming edges for each node
    3. Repeat until S is empty:
       3.1. Pick any node u from S
       3.2. Add u to the topological order
       3.3. For each (u, v_i), decrement W[v_i]
       3.4. If W[v_i] becomes 0, add v_i to S

Greedy

Proving Techniques

Exchange argument: Show that at each step, you can exchange S's current choice with G's current choice without hurting S's quality.

Example: Interval scheduling

1. Let G = $i_1, i_2, \ldots, i_k$ and S = $j_1, j_2, \ldots, j_m$ for an input L
2. Let P(m) be the proposition that if S returns m number of intervals, then G also returns m number of intervals

3. Base case: P(1). The optimal solution has only 1 interval. Trivially, G can pick any interval, hence P(1) is true
4. Inductive hypothesis: P(m) is true
5. $f(i_1) \le f(j_1)$ since G always chooses the request with the earliest finish time.
6. Therefore, S* = $i_1, j_2, \ldots, j_{m+1}$ is also an optimal solution (explain)

7. S** = $j_2, j_3, \ldots, j_{m+1}$ must be optimal for L\{i_i} for S* to be optimal for L. S** outputs m intervals
8. By construction, G outputs $i_2, \ldots, i_k$ for L\{i_i}. By the inductive hypothesis, G must output m schedules. Hence, m = k - 1
9. Hence, k = m + 1. Therefore, P(m+1) is true.

Structural bound: every possible solution must adhere to some min/max and show that G produces min/max

"Greedy stays ahead": Show that at each step, G is always as good as S. Show that "Greedy stays ahead" implies optimality

Interval Scheduling
- Rule: Schedule the request with the <u>earliest finish time</u>

```
IntervalScheduling(R):
  A ← []
  visited[s] ← true
  while R is not empty do
    choose r_i in R with earlier finish time
    add r_i to A
    delete r_j in R that are incompatible with r_i
  endwhile
  return A
```

<u>Pf. (Greedy Stays Ahead)</u>
1. Let $i_1, \ldots, i_k$ be the set of requests from G and $j_1, \ldots, j_m$ be the set of requests from S
2. It suffices to show that if $f(i_r) \le f(j_r)$ for all r ≤ k, then k ≥ m
3. Proof by induction on r
   1. Base case: r = 1. G will choose i_1 which is the request with earliest finish time
   2. Inductive hypothesis: Suppose $f(i_{r-1}) \le f(j_{r-1})$
   3. $f(i_{r-1}) \le f(j_{r-1})$, so $f(i_{r-1}) \le s(j_r)$
   4. Hence, j_r must be available for selection by G for the rth request
   5. Hence, $f(i_r) \le f(j_r)$

Interval Partitioning
- Rule: Consider the resources in the order of their <u>start time</u>

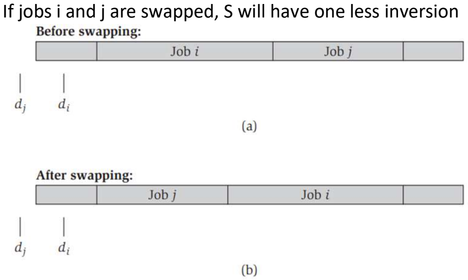
```
IntervalPartitioning (R):
  d ← 0
  sort intervals in R in ascending order of start time
  for j = 1 to n
    if interval j is compatible with some resouce k
      schedule interval j to resource k
    else
      allocate new resource d + 1
      schedule interval j to resource d + 1
      d ← d + 1
```

<u>Pf. (Structural Bound)</u>
- Observation: In any instance of interval partitioning, the number of resources needed is at least the depth of the set of intervals

Minimising Lateness
- Rule: Schedule requests with the earliest deadlines first

<u>Pf. (Exchange Argument)</u>
- Observation: An inversion is when job j is scheduled after job i when d(j) < d(i). If S has an inversion, then there must be a pair of jobs i and j such that j is scheduled immediately after i and has d(j) < d(i).

1. Suppose S has at least one inversion. Let job i be scheduled immediately after job j even though d(j) < d(i)
2. If jobs i and j are swapped, S will have one less inversion



3. Denote L' as the max lateness after swap and L as the max lateness before swap. Denote t(m) as the time taken to complete a job m
4. L' = max{ t(j) – d(j), t(j) + t(i) – d(i) }
   L = max{ t(i) – d(i), t(i) + t(j) – d(j) } = t(i) + t(j) – d(j), since d(j) < d(i)
5. t(j) – d(j) < t(i) + t(j) – d(j) and t(j) + t(i) – d(i) < t(i) + t(j) – d(j). Therefore, L' must be smaller than L.
6. We've shown that the lateness of S does not increase after the swap
7. This shows that an optimal schedule with no inversions exists.
8. All schedules with no inversions have the same maximum lateness (to proof). Hence, the schedule obtained by the greey solution is optimal.

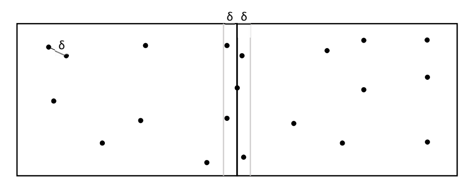
Divide and Conquer

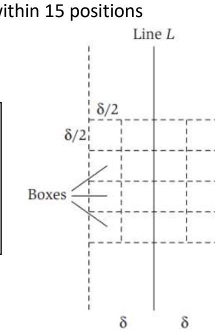
Counting Inversions
- Mergesort, but count the number of inversions during the merge step
- Key property: if L[i] > R[j], then L[i:] > R[j]

```
Merge (L, R):
  count ← 0
  i, j ← 0
  M ← empty array
  while i < len(L) and j < len(R) do
    if L[i] < R[j] then
      add L[i] to back of M and increment i
    else
      add R[j] to back of M and increment j
      increment count by len(L) – i + 1
  end while
  … regular merge steps
```

Finding the closest pair of points
- Find closest pair in left half L and right half R and those in the boundary split

1. Keep a sorted list P_x of points where points are sorted by x-axis.
   Keep a sorted list P_y of points where points are sorted by y-axis.
2. Split the points into two halves Q and R by their x-axis.
3. Recurse on both Q and R and find the minimum distance δ between two pairs of points between the two
4. Choose a point q in Q that has highest x value x* and draw a vertical line L through it. L essentially divides the points in Q and R
5. (**Pf 1) Narrow down to only points with x-values inside the [x* - δ, x* + δ] boundary.
6. Construct S_y containing only the points above, sorted by y-axis (can be done in O(n) using P_y)
7. Divide the [x* - δ, x* + δ] boundary into many δ/2 x δ/2 boxes
8. (**Pf 2)If there exists any 2 points p and p' such that d(p, p') < δ, then p and p' must be at most <u>**15**</u> positions away in S_y
9. Compute each d(p, p') in S_y such that p and p' are within 15 positions away and find the minimum d(p, p') = δ'
10. Return min { δ, δ' }





- **Pf 1**: If ∃ q = (q_x, q_y) ∈ Q and r = (r_x, r_y) ∈ R s.t. d(q,r) < δ, then q and r lies within distance δ of L
  $x^* - q_x \le r_x - q_x \le d(q,r) \le \delta$ and $r_x - x^* \le r_x - q_x \le d(q,r) \le \delta$
  → Hence q and r have x-coordinate within δ of L
- **Pf 2**: The max distance within a box is the length of the diagonal, which is $\sqrt{2\left(\frac{\delta}{2}\right)^2} = \sqrt{\frac{\delta^2}{2}} = \frac{\delta}{\sqrt{2}} \le \delta$. Hence, no two points can be in the same box
  Suppose there exists points s and s' in S_y such that d(s, s') < δ and that they are 16 positions apart. Assume WLOG that $s_y < s_y'$. Then, s and s' must be separated by at least 3 rows of boxes which must have a distance of at least $\frac{3}{2}\delta \ge \delta$ – a contradiction

Proving Techniques

Proof by (strong) induction on the input size *n*
1. Define the proposition
2. Show how the base case is fulfilled by the algorithm
3. Suppose P(k) is true for all k < n
4. By inductive hypothesis, P(n/c) must be true
5. Show how combining the subproblems causes P(n) to be true

Dynamic Programming

Knapsack Problem

$$DP(S,W) = \begin{cases} 0, & S = \emptyset, W \le 0 \\ max\{ w_i + DP(S\backslash\{n_i\}, W - w_i), DP(S\backslash\{n_i\}, W) \}, & S \ne \emptyset, W > 0 \end{cases}$$

```
Knapsack(Sn, W):
  M ← (n + 1) x (w + 1) array
  M[0][W] ← 0 for all w
  for i from 1 to n
     for w from 0 to W
        if wi < w then
           M[i][w] ← max{ wi + M[i - 1][w - wi], M[i - 1][w]}
        else
           M[i][w] ← M[i - 1][w]
  return M[n][W]
```

Network Flow

Definitions

- **s-t cut**: partition (A, B) of V such that s ∈ A and t ∈ B
- **cap(A, B)**: capacity of an s-t cut (A, B) = $\sum_{e\ out\ of\ A} c(e)$
- **s-t flow** must satisfy the following contraints:
  1. $0 \le f(e) \le c(e), \forall e \in E$ (capacity)
  2. $f^{in}(v) = f^{out}(v), \forall v \in V\backslash\{s,t\}$ (conservation)
- **Flow value** : $v(f) = f^{out}(s)$

Max-flow Min-cut Theorem (Ford-Fulkerson Algorithm)

- Max flow value from s to t = minimum capacity of any cut
- **Flow value lemma**: $f^{out}(A) - f^{in}(A) = v(f)$
  - The net flow sent across any cut in G = flow amount leaving s
- Weak duality: $v(f) \le cap(A,B)$ for any cut (A, B)
- Certificate of Optimality: Let f be any flow and (A, B) be any cut. If $v(f) = cap(A,B)$, then f is a max flow and (A, B) is a min cut
- with the smallest capacity will be the bottleneck edge *b*
- For every edge e along *P*, if e is a forward edge, we can increase $f(e)$ by $c(b)$. If e is a backward edge, we decrease $f(e)$ by $c(b)$

```
FordFulkerson(V, E):
   f(e) ← 0  ∀e ∈ E                  // initialise all flows to 0
   while ∃ s-t path in Gf do
      Find simple s-t path P in Gf (BFS/DFS)
      f ← Augment(f, P, G)
      Update Gf
   return f

Augment(f, P, G):
   b ← c(e) of bottleneck edge e along P
   for each e in P
      if e is a forward edge then
         increment f(e) in G by b
      else
         decrement f(e) in G by b
   return f + b
```

- **Residual Graph** G_f of G can be constructed using the following rules:
  1. Vertices in G_f and G are the same
  2. For each edge e of G, if $f(e) < c(e)$, then add e to G_f but with (residual) capacity = $c(e) - f(e)$
  3. For each edge e of G, if $f(e) > 0$, then add e to G_f but reverse the direction
- **Augmenting Path** is a simple path *P* in G_f from s to t.

Proof of Ford-Fulkerson

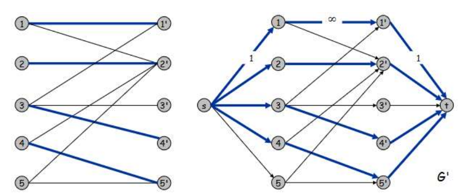
- Show TFAE
  1. There exists a cut (A, B) s.t. v(f) = cap(A, B)
  2. Flow f is a max flow
  3. There is no augmenting path relative to f
- 1 => 2: Corollary to weak duality lemma
- 2 => 3: Proof by contrapositive
  - If there exists an augmenting path, then we can improve f by sending flow along that path (assuming capacity is a non-negative integer)
- 3 => 1:
  1. Let f be a flow with no augmenting paths
  2. Let A be the set of vertices reachable from s in G_f and let B = V - A
  3. By defn of A, s ∈ A and t ∉ A (t ∈ B)
  4. Show that $f^{out}(A) = \sum_{e\ out\ of\ A} c(e)$
     a. There are no directed edges from A to B to some node u in B in G_f, otherwise vertex u would have been in set A
     b. Hence, if there is an edge e <u>from A to B</u> in G (G now, not G_f), f(e) = c(e) for e to not remain in G_f. i.e.
  5. Show that $f^{in}(A) = 0$
     a. Suppose $f^{in}(A) > 0$. Then, there exists an edge e(v,u) from B to A such that v ∈ B and u ∈ A.
     b. Then, there must have been a reverse edge e'(u,v) in G_f from A to B. If that's the case, v must have been reachable from s → Contradiction
  6. By Flow value lemma, $v(f) = f^{out}(A) - f^{in}(A)$
  7. Hence, $v(f) = \sum_{e\ out\ of\ A} c(e) = \mathrm{cap(A,B)}$ as shown above

Bipartite Matching

- Bipartite matching = set of edges s.t. no two edges in the set share the same endpoint. Maximum matching = largest of such set

<u>Max flow implementation</u>

- Add source s and join s to each node in L with edge of capacity = 1
- Add sink t and join each node in R to t with edge of capacity = 1
- Edges between L and R have infinite capacity
- Max flow in this graph G' = size of maximum matching in G



<u>Proof</u> (show k ≤ f and k ≥ f)

Show k ≤ f:

1. Given bipartite graph G with max matching of k.
2. Consider a flow f that sends 1 unit along each of the k paths
3. f is a flow and has cardinality k

Show k ≥ f:

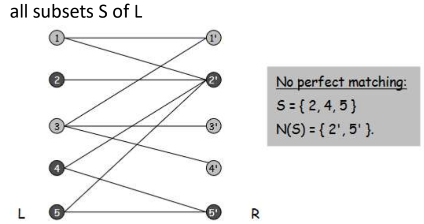
4. Let f be a max flow in G' with value = k
5. By integrality theorem, flow across any edge in G' is either 0 or 1
6. Each node in L has at most one incoming edge from s, hence its incoming flow is at most 1 and outgoing flow (to R) is at most 1
7. Each node in R has at most one outgoing edge to t, hence its outgoing flow is at most 1 and its incoming flow (from L) is at most 1
8. Hence, there must be a bipartite matching between L and R with cardinality k

Perfect Matching

- Perfect matching = bipartite matching which covers all the nodes
- Condition: |L| = |R|

<u>Marriage (Hall's) Theorem</u>

- A bipartite graph G = (L ∪ R) has a perfect matchinig iff |N(S)| ≥ |S| for all subsets S of L



<u>Proof of Marriage Theorem</u>

⇒: G has perfect matching →|N(S)| ≥ |S|

Each node in S needs to be mapped to a different node in N(S) for all subsets S of L

⇐: |N(S)| ≥ |S| → G has perfect matching. Proof by contrapositive

1. Suppose G does not have a perfect matching
2. Create G' from G using the same method in bipartite matching and let (A, B) be a min cut in G'
3. Max flow in G' = size of max matching in G. Since size of max matching < |L| (since G has no perfect matching), cap(A, B) < |L|
4. Let L_A = L ∩ A, L_B = L ∩ B, R_A = R ∩ A
5. cap(A, B) = |L_B| + |R_A|
   a. The outgoing edges from cut A must have flow = 1 because cap(A, B) < |L|. Hence, no edge in G (from L to R, with infinite capacity) will be part of the set of edges across the cut
   b. The remaining edges are those from s to L or from R to t.
   c. Edges from s to L across the cut are in L_B. Edges from R to t across the cut are from R_A
6. As established in 5a, all neighbours of nodes in L_A must be within A, so |N(L_A)| ≤ |R_A|
7. |N(L_A)| ≤ |R_A| = cap(A, B) - |L_B| < |L| - |L_B| = |L_A|
8. Hence, |N(L_A)| < |L_A|. Choose S = L_A

Intractability

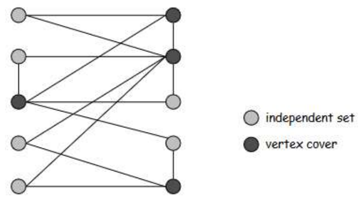
Polynomial-Time Reduction

- X polynomial reduces to Y if arbitrary instanced of X can be solved using polynomial number of calls to oracle that solved Y and polynomial number of standard computational steps (i.e. $X \le_p Y$)
- If X reduces to Y, then Y is at least as hard as X because if Y can be solved in polynomial time, then X can be solved in polynomial time
- **Intractability**: If $X \le_p Y$ and X cannot be solved in polynomial time, then Y cannot be solved in polynomial time
- **Equivalence**: If If $X \le_p Y$ and $Y \le_p X$, then $X \equiv_p Y$

Reduction by simple equivalence

e.g. Independent set $\equiv_p$ Vertex cover

- **Independent set** = Given graph G, is there a subset of vertices S such that |S|≥ k and there is no edge between each node in S
- **Vertex cover** = Given graph G, is there a subset of vertices S such that |S| ≤ k and for each edge, at least one of its endpoints is in S



Proof (S is an independent set iff V – S is a vertex cover)

⇒:

1. Let S be an independent set
2. Let (u, v) be an arbitrary edge
3. Since S is an independent set, u ∉ S or v ∉ S → u ∈ V – S or v ∈ V – S
4. Hence, V – S covers (u, v)

⇐:

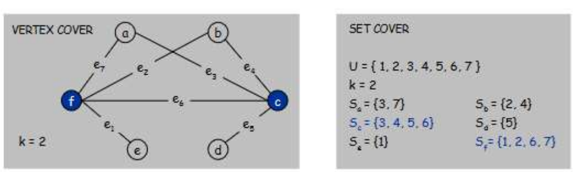
1. Let V - S be a vertex cover
2. Consider two nodes u ∈ S and v ∈ S (so u ∉ V - S and v ∉ V - S)
3. If u and v are joined by an edge (u, v), then either one of u or v must be in V – S, which contradicts (2).
4. Hence, no two nodes in S are joined by an edge → S is an independent set

- Therefore, ∃ Independent set ≥ k iff ∃ Vertex cover < k

Reduction from special case to general case

e.g. Vertex cover $\le_p$ Set cover

- **Set cover**: Given a set U of elements, a collection $S_1, S_2, \ldots, S_m$ of subsets of U, is there a collection of ≤ k of these sets whose union is equal to U



Proof (Vertex cover reduces $\le_p$ Set cover)

1. Construct specific set cover instance from graph G:
   a. Define set U to be set of all edges E in G and each subset of edges S_v = { e ∈ E: e is incident to v }
2. There exists a set cover of size ≤ k iff there exists a vertex cover of size ≤ k

Reduction via "gadgets"

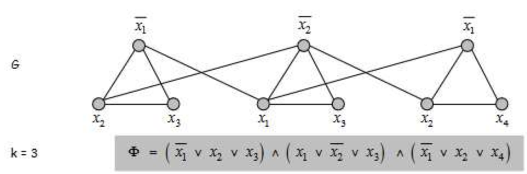
e.g. 3-SAT $\le_p$ Independent set

- **Definitions**:
  - Literal: Boolean variable or its negation $(x, \bar{x})$
  - Clause: Disjunction of literals $(x_1 \vee \overline{x_2} \vee x_3)$
  - Conjunction normal form (CNF, Φ) : Conjunction of clauses $(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_4 \vee x_5) \wedge (x_6 \vee \overline{x_7})$
- **3-SAT**: Given Φ where each clause contains exactly 3 literals, is there a satisfying truth assignment

Proof (3-SAT $\le_p$ Independent set)

Show G has an independent set S of |S|= k = |Φ| iff Φ of 3-SAT is satisfiable

1. Construct specific graph G
   a. G contains 3 vertices for each clause, one for each literal
   b. Connect the 3 literals in a clause in a triangle via an edge
   c. Connect literal to each of its negations via an edge
2. ⇒:
   - 2.1. Let S ⊆ G be an independent set of size k
   - 2.2. S must contain at least one vertex from each triangle because k = |Φ|
   - 2.3. S must contain at most one vertex from each triangle, otherwise there would be an edge between two vertices
   - 2.4. Hence, S must contain exactly one vertex from each triangle
   - 2.5. Furthermore, if u, v ∈ S, then u and v cannot be negations if each other otherwise there will be an edge (u, v) in G
   - 2.6. Set these literals in S to be true
   - 2.7. Then, the truth assignment will be consistent and all clauses are satisfied
3. ⇐:
   - 3.1. Given a satisfying assignment, at least one literal from each triangle must be true so that every clause is satisfied
   - 3.2. Pick one literal from each triangle to form an independent set S of size k



NP-completeness

Definitions

- **P**: Decision problem X with poly-time algorithms
- **NP**: Decision problems X with a poly-time certifier
  - Certifier C(s, t) returns 'yes' for some certificate |t| ≤ p(|S|)
- **NP-complete**: X ∈ NP and ∀ Y ∈ NP, $Y \le_p X$. Problem X is NP-complete if X is at least as hard as every NP problem

Properties

Proof: P ⊆ NP

1. Given X ∈ P, there exists a poly-time algorithm A that returns A(s).
2. Construct a certifier C(s, t) that returns A(s) and set t = ∅
3. Then, C(s, t) runs in poly-time and |t| ≤ p(|S|)

Proof: NP-complete problem X is solvable in polynomial time iff P = NP

⇒: X is in NP by definition. X is solvable in poly-time. Hence, NP ⊆ P

⇐: P = NP. X is in NP by definition. Hence, X is solvable in poly-time

Proof: If ∃ X ∈ NP s.t. X cannot be solved in polynomial time, then no NP-complete problem can be solved in polynomial time.

1. Contrapositive of the above proof: If P != NP, then no NP-complete problem is solvable in polynomial time

Polynomial Transformations

- **Cook** reduction: Problem X polynomial reduces to problem Y if X can be solved using (i) polynomial # of standard computational steps and (ii) polynomial # of calls to oracle that solves Y
- **Karp** reduction: Problem X polynomial transforms to problem Y if given any input x to X, we can construct an input y s.t. x is a 'yes' instrance of X iff y is a 'yes' instance of Y

Proof X is NP-complete

- If Y is NP-complete, X ∈ NP and $Y \le_p X$, then X is NP-complete:
  - Take an Z ∈ NP, then $Z \le_p Y \le_p X$. Hence, $Z \le_p X$.
  - By defn of NP-completeness, X is NP-complete
- Hence, it suffices to show:

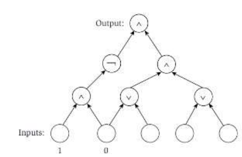
Show that X ∈ NP

Choose a known NP-complete problem Y

Prove $Y \le_p X$: (Karp reduction)

3.1. Construct an arbitrary instance s_Y of Y and construct instance s_X of X in polynomial time

3.2. Show that s_Y = 'yes' iff s_X = 'yes'



Circuit SAT $\le_p$ 3-SAT

- Circuit SAT: Known NP-complete problem
- Reduce any X ∈ NP to circuit SAT:
  - Set first n source nodes to be the n-bit s and remaining p(|s|) source nodes to represent the bits in t
  - X outputs 'yes' iff the ∃ t s.t. |t| ≤ p(|s|) and the circuit is satisfiable (i.e. C(s, t) = 'yes')
- Show 3-SAT is NP-complete by reducing Circuit SAT to 3-SAT

Construction (Circuit SAT ≤ₚ 3-SAT)

1. For each node v in Circuit SAT, let x_v be a variable in 3-SAT
2. Note that $P \Rightarrow Q = \neg P \vee Q$
3. NOT: $x_v \Leftrightarrow \overline{x_u} = (x_v \Rightarrow \overline{x_u}) \wedge (\overline{x_u} \Rightarrow x_v) = (\overline{x_v} \vee \overline{x_u}) \wedge (x_v \vee x_u)$
4. AND: $x_v \Leftrightarrow (x_u \wedge x_w) = \cdots = (\overline{x_v} \vee x_u) \wedge (\overline{x_v} \vee x_w) \wedge (x_v \vee \overline{x_u} \vee \overline{x_w})$
5. OR: $x_v \Leftrightarrow (x_u \vee x_w) = \cdots = (x_v \vee \overline{x_u}) \wedge (x_v \vee \overline{x_w}) \wedge (\overline{x_v} \vee x_u \vee x_w)$
6. SOURCE: Assign it to 0 or 1 if it is a hardcoded input
7. OUTPUT: Add variable $x_o = 1$ for output
8. Some clauses have < 3 variables. Create 4 new variables z_1, z_2, z_3, z_4 and add the clauses $(\overline{z_i} \vee z_3 \vee z_4), (\overline{z_i} \vee \overline{z_3} \vee z_4), (\overline{z_i} \vee z_3 \vee \overline{z_4}), (\overline{z_i} \vee \overline{z_3} \vee \overline{z_4})$ for each $i = 1, i = 2$. This ensures that $z_1 = z_2 = 0$
9. If a clauses has 1 variable t, replace it with $(t \vee z_1 \vee z_2)$
10. If a clause has 2 variables $(s \vee t)$, replace it with $(s \vee t \vee z_1)$

Proof

⇒ Suppose Circuit SAT is satisfiable

1. The satisfying assignment to Circuit SAT will create values at all nodes of the circuit
2. This set of values will satisfy the constructed SAT instance

⇐ Suppose 3-SAT is satisfiable

3. The clauses in 3-SAT ensure that the values assigned to all nodes of the circuit are the same as what the circuit computes for these nodes.
4. $x_o = 1$ in 3-SAT, so the assignment is satisfiable in Circuit SAT

3-SAT ≤ₚ Hamiltonian Cycle

Construction

1. Assume a 3-SAT instance Φ with n variables and k clauses,
2. For each variable x_i in 3-SAT, construct a path $P_i = v_{i,1}, v_{i,2}, \dots, v_{i,b}$ where $b = 3k + 3$. Edges in the path are bi-directional
3. For each path,
   - 3.1. Add edges from $v_{i,1}$ to $v_{i+1,1}$ and $v_{i+1,b}$
   - 3.2. Add edges from $v_{i,b}$ to $v_{i+1,1}$ and $v_{i+1,b}$
4. Add s and edges from s to $v_{1,1}$ and $v_{1,b}$
5. Add t and edges from $v_{n,1}$ and $v_{n,b}$ to t
6. This models a Hamiltonian Cycle with 2^n paths (each 'layer' can be traversed from L2R or R2L, independent of other 'layers'). Similarly, there are 2^n different assignments in a 3-SAT instance
7. If a path P_i is traversed from L2R, set $x_i = 1$. Else $x_i = 0$
8. Add an extra node c_j for each clause C_j
   - 8.1. For each variable x_i in C_j, add the edges $x_{i,3j} \to c_j$ and $c_j \to x_{i,3j+1}$ if x_i is not a negation. Otherwise, add the edges $x_{i,3j+1} \to c_j$ and $c_j \to x_{i,3j}$
   - 8.2. e.g. if $C_1 = x_1 \vee \overline{x_2} \vee x_3$, P_1 must go from L2R <u>OR</u> P_2 must go from R2L <u>OR</u> P_3 must go from L2R in order for c_j to be visited

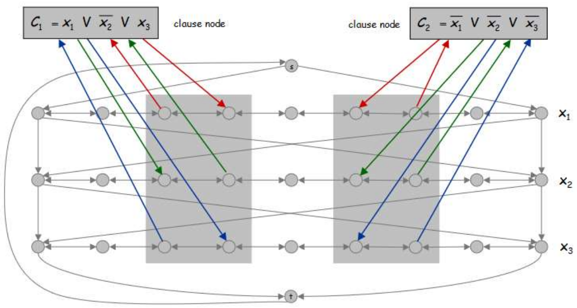
Proof

⇒ Suppose Φ is satisfiable

1. If an arbitrary variable x_i is 1, then P_i traverses from L2R, otherwise it traverses from R2L.
2. For each clause C_j, since it is satisfied, there must be a path P_i that traverses in the "correct" direction so c_j can be spliced into the cycle via edges incident on $v_{i,3j}$ and $v_{i,3j+1}$

⇐ Suppose there exists a Hamiltonian Cycle $\mathfrak{C}$

3. Then all c_j must be visited
4. If $\mathfrak{C}$ enters a node c_j from $v_{i,3j}$, it must immediately depart on an edge to $v_{i,3j+1}$ otherwise $v_{i,3j+2}$ will never be visited without breaking the Hamiltonian property
5. Symmetrically, if $\mathfrak{C}$ enters a node c_j from $v_{i,3j+1}$, it must immediately depart on an edge to $v_{i,3j}$ otherwise $v_{i,3j-1}$ will never be visited without breaking the Hamiltonian property
6. Hence, for each c_j, the nodes before and after c_j in $\mathfrak{C}$ are joined by an edge in G.
7. Therefore, we can remove each c_j in $\mathfrak{C}$ and join $v_{i,3j}$ and $v_{i,3j+1}$ via an edge, forming a Hamiltonian cycle $\mathfrak{C}'$ of $G' = G - \{c_1, \dots, c_k\}$
8. Any Hamiltonian cycle in G' must traverse each P_i in only one direction. Hence, each P_i in $\mathfrak{C}'$ must traverse fully in only one direction
9. If $\mathfrak{C}'$ traverses P_i from L2R, we can set $x_i = 1$, otherwise $x_i = 0$
10. Since the larger cycle $\mathfrak{C}$ was able to visit all c_j, at least one of the paths was traversed in the correct direction relative to c_j
11. Hence, the truth assignment satisfies all the clauses.



Hamiltonian Cycle ≤ₚ Traveling Salesman Problem

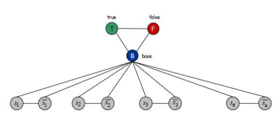
Construction

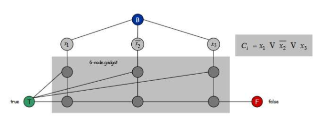
1. Given graph G(V, E), create |V| cities with distance function $d(u,v)\begin{cases} 1, & (u,v) \in E \\ 2, & (u,v) \notin E \end{cases}$
2. Then, G will have a TSP tour of length ≤ |V| iff G is Hamiltonian

3-SAT ≤ₚ 3-Colorability

Construction

1. For each variable x_i, create nodes x_i and $\overline{x_i}$ and join them via an edge
2. Create 3 extra nodes, T, F and B that connect to one another. Connect B to every literal
3. For each clause, create a 6-node gadget:





4. Φ is satistiable iff the constructed graph is 3-colorable

Proof

⇒ Suppose Φ is satisfiable

1. Color all true literals green
2. Color nodes below green nodes to be red and the node below blue
3. Color remaining middle row nodes blue (first layer of nodes in gadget)
4. Color remaining bottom nodes red/green as forced
5. The resulting graph is 3-colorable

⇐ Suppose the graph is 3-colorable

6. Assign each literal coloured green to be true
7. Each variable must have one literal to be green and the other to be red (refer to left image). Hence, the assignment is consistent
8. No literals can be blue, hence each literal is assiged T or F
9. At least one literal in any clause will be true (green) to satisfy 3-colorability (refer to right graph)
10. Hence, Φ is satisfiable

3-SAT ≤ₚ Subset-Sum

Construction

1. Given Φ with n variables and k clauses, create 2n + 2k integers, each with n + k digits
2. For each integer, the first n digits represent each variable and the last k digits represent each clause
3. For each variable x_i, create 2 integers for x_i and $\overline{x_i}$. For both integers, set digit i to 1. For each clause digit, set it to 1 if the literal is part of the clause.
4. For each clause c_j, create 2k integers. Set first n digits to be 0. Set digit j to 1 for one of the integers and 2 for the other.
5. Then, Φ is satisfiable iff there exists a subset that sums to W = 11…144…4 (n 1s and k 4s)

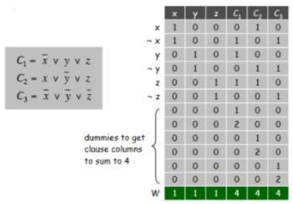
Proof

⇒ Suppose Φ is satisfiable

1. For each variable x_i, choose one of the two integers representing it (depending on where x_i is T or F)
2. There will be n of such integers.
3. For each clause c_j, choose one or both of the two integers representing it such that the digit j of the resulting sum is 4
4. Sum the n + k digits to get S
5. The first n digits of S will be 1 because exactly one integer is picked for each integer
6. The last k digits of S will be 4; because Φ is satisfiable, digit j has value 1, 2 or 3 from the sum of the n integers in (2). We can always pick some 1 or 2 of the dummy rows to make it up to 4
7. Take the selected integers. Then, there exists a subset sum equals to W

⇐ Suppose there exists a subset sum equals to W

8. Let the subset of integers that sum to W be S
9. Then, exactly one of the integers in S must have digit $i \leq n$ set to 1. To achieve this, exactly one of the literals for each variable in Φ must be true (therefore, there is consistent assignment)
10. Each clause c_j is true in order for each digit $j \geq k$ to be 4
11. Hence, Φ is satisfied



$C_1 = \bar{x} \vee y \vee z$
$C_2 = x \vee \bar{y} \vee z$
$C_3 = \bar{x} \vee \bar{y} \vee \bar{z}$

	x	y	z	C_1	C_2	C_3
x	1	0	0	0	1	0
¬x	1	0	0	1	0	1
y	0	1	0	1	0	0
¬y	0	1	0	0	1	1
z	0	0	1	1	1	0
¬z	0	0	1	0	0	1
dummies to get clause columns to sum to 4	0	0	0	1	0	0
	0	0	0	2	0	0
	0	0	0	0	1	0
	0	0	0	0	2	0
	0	0	0	0	0	1
	0	0	0	0	0	2
W	1	1	1	4	4	4

Approximation Algorithms

Load Balancing (2-approx)

- Given m machines and n jobs, each job j with processing time t_j. Each job must run contiguously on one machine and each machine processes at most one job at a time. Load $L_i = \sum t_i$ for each machine. Assign the jobs such that maximum load on any machine (makespan) is minimised
- Greedy solution: Assign job j to machine i with the smallest load so far. Then, the makespan $L \leq 2L^*$ where L^* is the optimal makespan
- Optimisation: Sort the jobs in descending order of processing time. Then, the algorithm is a 1.5-approximation

Proof (w/o sorting)

1. $L^* \geq \max t_j$ since some machine must process the most time-consuming job
2. $L^* \geq$ average load $= \frac{1}{m}\sum_j t_j$
3. Let machine i be the machine with the maximum load L_i and job j be the last job assigned to it.
4. Then, machine i must have had the smallest load before this assignment. i.e. $L_i - t_j \leq L_k$ for any machine L_k
5. $L_i - t_j \leq \frac{1}{m}\sum_k L_k = \frac{1}{m}\sum_k t_k \leq L^*$
6. Hence, $L_i = (L_i - t_j) + t_j \leq 2L^*$

Proof (with sorting)

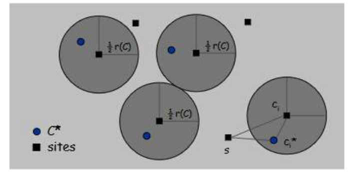
2. If there are ≤ m jobs, then the greedy solution is optimal (i.e. $L = L*$)
3. If there are > m jobs, consider the first $m+1$ jobs $t_1, t_2, \ldots, t_m + 1$
4. Each job takes at least t_{m+1} time since they are sorted in desc order
5. There are $m+1$ jobs and m machines. By pigeonhole principle, at least one machine gets 2 jobs.
6. Therefore, $L^* \geq 2t_{m+1}$
7. Hence, $L_i = (L_i - t_j) + t_j \leq L^* + \frac{1}{2}L^* = \frac{3}{2}L^*$

Center Selection Problem (2-approx)

- Given n sites, select k centers C so that the max distance $r(C)$ from a site to nearest center is minimised
- Greedy solution: Repeatedly choose the next center to be the site farthest away from any existing center (i.e. $dist(s, C)$ is the largest). Let first center to be at any arbitrary site. Then, $r(C) \leq r(C^*)$ where C^* is the set of optimal centers

Proof

1. Each iteration of the greedy algorithm reduces $dist(s, C)$ towards $r(C)$. Hence, upon termination, all centers in C are pairwise at least $r(C)$ apart
1. Assume (for contradiction) that $r(C) > 2r(C^*) \Rightarrow r(C^*) < \frac{1}{2}r(C)$
2. For each center c_i in C, consider a ball of radius $\frac{1}{2}r(C)$ around it
3. Because of (1), one ball will have exactly one c_i. Because of (2), one ball will have exactly one c_i^*
   - If ∃ a ball with no c_i^*, then $dist(c_i, C^*) > \frac{1}{2}r(C)$, a contradiction
   - If ∃ a ball with multiple c_i^*, then there will ∃ a ball with no c_i^*



4. Consider any site s and its closest center c_i^* in C^*
5. Then, $dist(s, C) \leq dist(s, c_i) \leq \underbrace{dist(s, c_i^*) + dist(c_i^*, c_i)}_{\Delta\text{-inequality}} \leq 2r(C^*)$, a contradiction
6. Hence, $r(C) \leq 2r(C^*)$

Randomized Algorithms

- **Monte Carlo**: Guaranteed poly-time, likely correct answer
- **Las Vegas**: Guaranteed correct answer, likely poly-time

Contention Resolution (Monte Carlo)

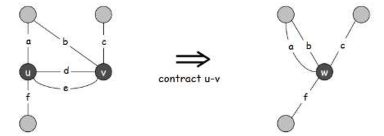
- Given n processes $P_1, P_2, \ldots, P_n$ and a single resource r which can only be accessed by at most one process at any given time, devise a protocol to ensure that all processes can access r on a regular basis
- Randomized Protocol: Each process requests access to r at time t with probability $p = 1/n$

Proof

1. Let $S[i,t]$ = the event that P_i successfully accesses r at time t
2. Then, $Pr(S[i,t]) = p(1-p)^{n-1} = \frac{1}{n}\left(1 - \frac{1}{n}\right)^{n-1}$
3. $\frac{1}{e} \leq \left(1 - \frac{1}{n}\right)^{n-1} \leq \frac{1}{2}$. Therefore, $\frac{1}{en} \leq S[i,t] \leq \frac{1}{2n}$
4. Let $F[i,t]$ = the event that P_i fails to access r in rounds 1 through t
5. Then, $Pr(F[i,t]) = \left(1 - Pr(S[i,t])\right)^t \leq \left(1 - \frac{1}{en}\right)^t$
6. Choose $t = \lceil en \rceil$. Then, $\Pr(F[i,t]) \leq \left(1 - \frac{1}{en}\right)^{\lceil en \rceil} \leq \left(1 - \frac{1}{en}\right)^{en} \leq \frac{1}{e}$
7. Choose $t = \lceil en \rceil \lceil c \ln n \rceil$. Then, $\Pr(F[i,t]) \leq \left(\frac{1}{e}\right)^{c \ln n} = n^{-c}$
8. Let $F[t]$ = the event that at least one of the n processes fails to access r in any of the rounds 1 through t
9. Then, $\Pr(F[t]) = \Pr(\bigcup_{i=1}^n F[i,t]) \leq \sum_{i=1}^n \Pr(F[t]) \leq n\left(1 - \frac{1}{en}\right)^t$
   - Union Bound: The probability that ≥ 1 of the events happens is at most the sum of the probabilites of the individual events
10. Choose $t = \lceil en \rceil \lceil 2 \ln n \rceil$. Then, $\Pr(F[t]) \leq n \cdot n^{-2} = \frac{1}{n}$
11. Therefore, the probability that all the processes succeeds to access r within rounds $2en \ln n$ rounds is at least $1 - \Pr(F[t]) = \frac{n-1}{n}$

Global Minimum Cut (Monte Carlo)

- Given a connected, undirected and unweighted graph G(V, E), find a cut (A, B) of minimum cardinality (least # edges across the cut)
- Contraction Algorithm:
  1. Pick any edge (u, v) at random
  2. Contract edge (u, v): Replace vertices u and v with new super-node w and remove all edges between u and v
  3. Repeat until graph has just two nodes v_1 and v_2
  4. Return the cut (all vertices that were contracted to form v_1)
  5. Then, the probability that this algorithm returns a global min-cut is $\geq 2/n^2$



Proof

1. Let (A^*, B^*) be the optimal global min-cut of and F^* be the set of edges across the cut. Let $|F^*| = k$
2. The first iteration of the contraction algorithm picks and contracts an edge in F^* with probability $\frac{k}{|E|}$
3. Suppose $\exists v \in V$ s.t. degree(v) < k. Take v to form A^* and the rest of the vertices to form B^*. Then, the global min-cut < k, a contradiction. Hence, $\forall v \in V$, degree(v) ≥ k.
4. For any graph, $|E| = \frac{1}{2}\sum_{v_i \in V}$ degree(v_i). Therefore, $|E| \geq \frac{1}{2}kn$
5. Hence, by point (2), the algorithm contracts an edge in F^* in the first iteration with probability $\leq \frac{k}{|E|} \leq \frac{k}{\frac{1}{2}kn} = \frac{2}{n}$
6. After j iterations, the number of vertices remaining is $n' = n - j$ (each iteration reduces the # vertices by one).
7. Suppose no edge in F^* were contracted in these j iterations. Then, the min-cut is still k. Hence, $|E'| \geq \frac{1}{2}kn'$ and the algorithm contracts an edge in F^* with probability $\leq \frac{2}{n\prime}$
8. Let E_j = event that an edge in F^* is not contracted in iteration j
9. Then, $Pr(E_1 \cap E_2 \cap \ldots \cap E_{n-2}) = Pr(E_1) \times Pr(E_2|E_1) \times Pr(E_3|E_1 \cap E_2) \times \ldots \times Pr(E_{n-2}|E_1 \cap E_2 \cap \ldots \cap E_{n-3}) \geq \left(1 - \frac{2}{n}\right)\left(1 - \frac{2}{n-1}\right)\left(1 - \frac{2}{n-2}\right)\ldots\left(1 - \frac{2}{4}\right)\left(1 - \frac{2}{3}\right) = \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right)\ldots\left(\frac{2}{4}\right)\left(\frac{1}{3}\right) = \frac{2}{n(n-1)} \geq \frac{2}{n^2}$
10. Run the algorithm $n^2 ln\, n$ times with independent random choices. Then, the probability of failing is

$$\leq \left(1 - \frac{2}{n^2}\right)^{n^2 \ln n} = \underbrace{\left(\left(1 - \frac{2}{n^2}\right)^{\frac{1}{2}n^2}\right)^{2\ln n}}_{1 + x \leq e^x \Rightarrow 1 - 1/x \leq e^{-1/x} \Rightarrow (1 - 1/x)^x \leq e^{-1}} \leq e^{-2\ln n} = \frac{1}{n^2}$$

Expectation

- $\boldsymbol{E[X] = \sum_{j=0}^{\infty} j \Pr(X = j)}$
- $\boldsymbol{X}$ = # trials until first success = $\boldsymbol{X \sim G(p) \Rightarrow E[X] = 1/p}$
- $\boldsymbol{X}$ = Bernoulli Trial = $\boldsymbol{X \sim Bernoulli(p) \Rightarrow E[X] = p}$
- Linearity of expectation (LoE): $\boldsymbol{E[X+Y] = E[X] + E[Y]}$

Max 3-SAT (Las Vegas)

- Given Φ with k clauses, the expected # of clauses satisfied by a <u>random</u> assignment is 7k/8
- Hence, there must exist a random assignment that satisfies at least 7k/8 clauses with probability ≥ 1/8k
- Johnson's algorithm: If we repeatedly generate random assignments until one satisfies ≥ 7k/8 clauses, the number of trials is ≤ 8k

<u>Proof</u>

1. Let X_j = RV that equals 1 if clause j is satisfied and 0 otherwise
2. Hence, $\mathrm{E}[X_j] = p = 7/8$ (Bernoulli Trial)
3. Let X = # of satisfied clauses i.e. $X \sim B(k, 7/8)$. Then, $\mathrm{E}[X] = \mathrm{E}[X_1 + X_2 + \cdots + X_k] = E[X_1] + E[X_2] + \cdots + E[X_k] = 7k/8$ (LoE)
4. Let $p_j = \Pr[\#\text{satisfied} = j]$ and $p = \Pr[\#\text{satisfied} \geq 7k/8]$ Hence,
$$7k/8 = E[X] = \sum_{j>0} j\,p_j = \sum_{j<7k/8} j\,p_j + \sum_{j\geq 7k/8} j\,p_j$$
$$\leq \left(\frac{7k}{8} - \frac{1}{8}\right) \sum_{j<7k/8} p_j + k \sum_{j\geq 7k/8} p_j$$
$$\leq \left(\frac{7k}{8} - \frac{1}{8}\right)(1) + kp$$
5. Therefore, $p \geq 1/8k$
6. Let Y = # trials until one assignment satisfies ≥ 7k/8 clauses. Then, $Y \sim G(p \geq 1/8k)$
7. Hence, $E[Y] \leq 1/(1/8k) = 8k$

Universal Hashing

- **Universal class of hash functions** = a set H of hash functions h_i s.t. $\Pr[h(u) = h(v)] \leq 1/n, \quad \forall \boldsymbol{u}, \boldsymbol{v} \in U$, n = #buckets
- Let X = # collisions with any $\boldsymbol{u} \in U$ using H. Then. for any $S \subseteq$ U and $|S| \leq 1/n$, $E_{h\in \mathrm{H}}[X] \leq 1$

<u>Proof</u>

1. Let Let X_v = RV that equals 1 if v collides with u and 0 otherwise
2. Hence, $E[X_v] = \Pr[h(u) = h(v)] \leq 1/n$
3. Let X = # collisions with any $\boldsymbol{u}$ i.e. $X \sim B(|S|, p \leq 1/n)$. Then, $\mathrm{E}[X] \leq |S|/n \leq 1$

- Designing such a universal class:

1. Choose a prime number p, $n \leq p \leq 2n$ (Chebyshev: such a p exists)
2. For each $\forall \boldsymbol{u} \in U$, identify a base-$p$ integer of r digits: $x = (x_1, x_2, \dots, x_r)$
3. Let A be the set of all possible r-digit, base-p integers i.e. for each $\boldsymbol{a} \in A$, $a = (a_1, a_2, \dots, a_r), 0 \leq a_i < p$
4. Then, $H = \{\, h_a \mid h_a(x) = (\sum_{i=1}^{r} a_i x_i)\ mod\ p, a \in A \,\}$

<u>Proof</u>

1. Let $x = (x_1, x_2, \dots, x_r)$ and $y = (y_1, y_2, \dots, y_r)$, $x \neq y$
2. Then, $\exists j$ s.t. $x_j \neq y_j$
3. $h_a(x) = h_a(y) \Leftrightarrow \mathbf{a_j}\overbrace{(\mathbf{y_j} - \mathbf{x_j})}^{z} = \overbrace{\sum_{i\neq j} \boldsymbol{a_i}(\boldsymbol{x_i} - \boldsymbol{y_i})}^{m} \bmod \boldsymbol{p}$
4. Assume vector a is chosen randomly by first (uniformly) randomly picking each a_i , $i \neq j$ in the range [0, p], then picking a_j at random
5. Since p is prime, $a_j z = m \bmod p$ has at most (exactly) 1 solution among p possibilities

1. Let p be prime, $z \neq 0 \bmod p$ (z not divisible by p).
2. Suppose α and β are 2 different solutions (for contradiction)
3. Then, $\alpha z = m + k_1 p$ and $\beta z = m + k_1 p$. So, $(\alpha - \beta)z = (k_1 - k_2)p = 0 \bmod p$
4. Hence, $(\alpha - \beta)$ is divisible by p since z not divisible by p
5. Therefore, $\alpha = \beta$ since $0 \leq \alpha, \beta < p$ (contradiction)
6. Hence, $\Pr[h_a(x) = h_a(y)] \leq 1/p \leq 1/n$ since $n \leq p \leq 2n$

Chernoff Bounds

- Let $X_1, X_2, \dots, X_n$ be a Bernoulli process (i.e. they are <u>independent</u> 0-1 RVs) and $X = X_1, X_2, \dots, X_n$. Then, $\forall \mu \geq E[X]$ and $\forall \delta > 0$,
$$\Pr[X > (1+\delta)\mu] < \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu$$
and $\forall \mu \leq E[X]$ and $\forall \delta \in (0,1)$,
$$\Pr[X < (1+\delta)\mu] < e^{-\delta^2 \mu/2}$$

<u>Proof</u> (above mean)

1. Markov's inequality states that $\Pr[X > a] \leq E[X]/a$
2. Hence, $\Pr[X > (1+\delta)\mu] = \Pr\left[e^{tX} > e^{t(1+\delta)\mu}\right] \leq e^{-t(1+\delta)\mu} E[e^{tX}]$
3. $E[e^{tX}] = E\left[e^{t\sum_i X_i}\right] = E[e^{tX_1} e^{tX_2} \dots e^{tX_n}] = \prod_i E[e^{tX_i}]$ (independence: $X \perp Y \Rightarrow E(XY) = E(X)E(Y)$)
4. Let $p_i = \Pr[X_i = 1]$. Then,
$$E[e^{tX_i}] = p_i e^t + (1 - p_i)e^0 = \underbrace{1 + p_i(e^t - 1) \overset{1+x\leq e^x}{\leq} e^{p_i(e^t - 1)}}$$
5. Therefore, $\Pr[X > (1+\delta)\mu] \leq e^{-t(1+\delta)\mu} \prod_i E[e^{tX_i}] \leq e^{-t(1+\delta)\mu} \prod_i e^{p_i(e^t-1)} \leq e^{-t(1+\delta)\mu}\, e^{\mu(e^t - 1)}$ $(\mu \geq E[X] = \sum_i p_i)$
6. Take $t = \ln(1+\delta)$. Then,
$$\Pr[X > (1+\delta)\mu] \leq e^{-\ln(1+\delta)(1+\delta)\mu} e^{\mu(e^{\ln(1+\delta)} - 1)} = \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu$$

Load Balancing

- $\boldsymbol{m}$ jobs arrive in a stream and need to be processed immediately by n identical processors. Assign jobs to processors uniformly at random without centralised controller (without round-robin, where each processor will receive at most $\lceil m/n \rceil$ jobs. How likely is it that some processor is assigned "too many" jobs?

<u>Analysis</u>

1. Let X_i = # jobs assigned to processor i
2. Let Y_{ij} = RV that equals 1 if job j is assigned to processor i
3. Then, $E[Y_{i,j}] = 1/n$ $(E(X) = p, if\ X \sim Bernoulli(p))$
4. Hence, $X_i = \sum_j Y_{ij} \Rightarrow \mu = E[X_i] = nE[Y_{ij}] = 1$
5. By Chernoff bounds, with $\delta = c - 1$, $\Pr[X_i > c] < \frac{e^{c-1}}{c^c}$ for some c which will be chosen later
6. Let $\gamma(n) = x$ s.t. $x^x = n$ and choose $c = e\gamma(n)$
7. Then, $\Pr[X_i > c] < \frac{e^{c-1}}{c^c} < \left(\frac{e}{c}\right)^c = \left(\frac{1}{\gamma(n)}\right)^{e\gamma(n)} < \left(\frac{1}{\gamma(n)}\right)^{2\gamma(n)} = \frac{1}{n^2}$
8. By Union Bound, $\Pr[\cup_{i=1}^n X_i > c] \leq n\left(\frac{1}{n^2}\right) = \frac{1}{n}$. That is, the probability that at least one processor received more than $c = e\gamma(n)$ jobs is $\frac{1}{n}$.
9. Therefore, the probability that no processors received more than $c = e\gamma(n) = \Theta(\text{logn}/\log\text{logn})$ jobs is $1 - \frac{1}{n}$

- Suppose there are $m = 16n\ln n$ jobs. Then, $E[X_i] = 16\ln n$. With high probability, every processor will have betweeen half and twice the average load (e.g. $8\ln n = \frac{1}{2}\mu \leq X_i \leq 2\mu = 16\ln n$)

<u>Proof</u>

1. By Chernoff bounds, with $\delta = 1$,
$$\Pr[X_i > 2\mu] < \left(\frac{e}{4}\right)^{16\ln n} < \left(\frac{1}{e}\right)^{2\ln n} = \frac{1}{n^2}$$
$$\Pr\left[X_i < \frac{1}{2}\mu\right] < e^{-\frac{1}{2}\left(\frac{1}{2}\right)^2 (16\ln n)} = \frac{1}{n^2}$$
2. By Union Bound, $\Pr[\cup_{i=1}^n X_i > 2\mu] \leq n\left(\frac{1}{n^2}\right) = \frac{1}{n}$ and $\Pr\left[\cup_{i=1}^n X_i < \frac{1}{2}\mu\right] \leq n\left(\frac{1}{n^2}\right) = \frac{1}{n}$.
3. Therefore, the probability that every processor has load betweeen half and twice the average load is ≥ $1 - \frac{1}{n} - \frac{1}{n} = 1 - \frac{2}{n}$