COMS 311 FINAL EXAM REVIEW

Link to the 2nd Midterm Review for anyone who needs it

Have fun 😇

FRIENDLY REMINDER THE EXAM IS Comprehensive (just found out there's a difference)

1. Let $A = [a_1, \cdots a_n]$ be an array of positive and negative integers. Find i and j such that $1 \leq i \leq j \leq n$ and such that

$$\sum_{k=i}^{j} a_k$$

is maximized. Design a dynamic programing algorithm (without memoization/recursion) for this problem. State the recurrence relation.

Ans:

https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/ Here is an algorithm by GeeksForGeeks.

For a_k we have 2 possible outcomes: either a_k is in the maximized subarray or it's not. If a_k is in the subarray S then S is a maximized subarray at ... + a_{k-1} + $\mathbf{a_k}$. If a_k is *not* in the subarray S, then S is maximized subarray at $\mathbf{a_{k-1}}$. The recurrence relation we can derive for S[k] (or the maximum value of a subarray at position n) is:

Recurrence = Opt (k) = max {a_k, a_k+Opt(a_{k-1})} or

$$S[k] = Max\{S[k-1] + A[k], A[k]\}$$

Now that we have our recurrence relation, we can develop a dynamic programming iterative algorithm to find the maximum sum of a subarray.

```
MaxValueSubArray(A) {
      S = Array of size n   // Size of A
      S[0] = A[0]
      maxArrayI = 0
      maxArrayJ = 0
      maxValue = S[0]
      For k = 1 to n {
            S[k] = Max(S[k - 1] + A[k], A[k])

            If A[k] > S[k - 1] + A[k] && maxValue < A[k] {
                  set maxArrayI and maxArrayJ = k
                  maxValue = A[k]
            }
            Else if maxValue <= S[k] {
                  maxArrayJ = k
                  maxValue = S[k]
            }
      }
```

```
      Return maxArrayI and maxArrayJ

      // Another potential solution to keep track of indices is to just
      work back through S from n to 0 and get the highest value array
      rather than trying to keep track of it during calculations
}
```

```
Java implementation of algo above for testing.
static void solve(int[] input) {
      int i, j, k, max;
      int[] S = new int[input.length];

      S[0] = input[0];
      i = 0;
      j = 0;
      max = S[0];

      for (k = 1; k < S.length; k++) {
            S[k] = Integer.max(S[k - 1] + input[k], input[k]);
            if (input[k] > S[k - 1] + input[k] && max < input[k]) {
                  i = k;
                  j = k;
                  max = input[k];
            }
            else if (max <= S[k]) {
                  j = k;
                  max = S[k];
            }
      }
      System.out.println("Max Sum: " + S[j]);
      System.out.println("Index Range: " + i + " -> " + j);
}

public static void main(String[] args) {
      // solve(new int[] { 1, 2, -3, -4, 2, 7, -2, 3 });
      solve(new int[] { 1, 2, -3, -4, 2, -2 });
}
```

Here is a corrected set of python code:

```
def mySolution(A):
  i=0
  j=0
  maxV = -255
  S = [0] * len(A)
  S[0] = A[0]

  for k in range(1, len(A)):
    S[k] = max(S[k-1] + A[k], A[k])

    if S[k-1] + A[k] > A[k] and S[k] > maxV:
      j = k
```

```
            maxV = S[k]

        elif A[k] >= maxV:
            i = k
            j = k
            maxV = A[k]


    print('Range: %d->%d, Max: %d' % (i, j, maxV))


def main():
    A = [1, 3, -3, -5, -7, 1, -1, 4, -2]
    mySolution(A)


if __name__ in "__main__":
    main()
```

This takes O(n) time.

2. Given an undirected graph $G = (V, E)$, a subset S of V is called a *vertex cover* if for every edge $\langle u, v \rangle \in E$, at least one of u or v belongs to S. Let T be a binary tree. Give a dynamic programming algorithm (without memoization/recursion) that computes the smallest size vertex cover of T. State the recurrence relation.

Ans:

So for root node v we have 2 outcomes. Either v is in the vertex cover or it isn't. If v is in the vertex cover, then that means its children may or may not be part of the vertex cover. If v is not in the vertex cover, then that means its children are part of the vertex cover. What we can do is create a 2D matrix M of size N X 2 where N is the size of T. M[v][0] will be the size of the vertex cover if v is NOT included and M[v][1] will be the size of the vertex cover if v IS included.

Using these as our baseline we get our recurrence relation:

For M[v][0]:

$$M[v][0] = \sum_{c \in child(v)} M[c][1]$$

For M[v][1]:

$$M[v][1] = 1 + \sum_{c \in child(v)} Min(M[c][1], M[c][0])$$

Now that we have our recurrence relation, we can derive an algorithm:

```
MinSizeVertexCover(T) {

      M = 2D array of size N X 2
      Let T' be every leaf in T
      For each vertex x in T' {
            M[x][0] = 0
            M[x][1] = 1
      }

      For each vertex v in T -
T' {
            M[v][1] = 1
            For every child c in v {
                  M[v][0] += M[c][1]
                  M[v][1] += Min(M[c][1], M[c][0])
            }
      }
      Return Min(M[N][0], M[N][1])
```

Runtime should be O(N + M)

Answer 2: Use BFS and keep track of levels. Have 2 sets RinS and RninS (root in S, root not in S). When you mark a node visited and record the level, add vertext to the appropriate set base on levels[curNode] % 2 != 0. Return the min(|RninS|,|RinS|).

3. Given an undirected graph $G = (V, E)$, a subset S of V is called an *independent set* if for every $x, y \in S$, there is no edge from x to y in G.

   Let $G = (V, E)$ be a graph such that $V = \{v_1, v_2, \cdots, v_n\}$ and $E = \{\langle v_i, v_{i+1}\rangle \mid 1 \leq i < n\}$. (That is, G is a "line graph".) Assume that each vertex v has a non-negative weight $w(v)$. Given a set S of vertices, the weight of S is

$$\sum_{v \in S} w(v).$$

   Give a dynamic programming based algorithm (without memoization/recursion) to compute a maximum weight independent set for a line graph G. State the recurrence relation.

Ans:
https://www.geeksforgeeks.org/maximum-sum-such-that-no-two-elements-are-adjacent/ Here is an algorithm by GeeksForGeeks that can help.

Similar to the above problem, for the final vertex v there are two options. Option 1 is that v is in the independent set, and Option 2 is that v is not in the independent set. If v is in the independent set, then the weight of S is $\sum_{u \in S} w(u)$ (up to v - 2) + w(v). If v is not in the independent set, then the weight of S is $\sum_{u \in S} w(u)$ (up to v - 1). Let's create an array W of size n. Then, at position v, the recurrence relation would be:

$$W[v] = Max(W[v-2] + w(v),\ W[v-1])$$

Using this recurrence relation, we can derive an algorithm to find the maximum weight of an independent set:

```
maxWeightIndependentSet(G = (V, E)) {
      Let W be an array of size N
      W[1] = w(v1)
      W[2] = Max(W[1], w(v2))

      For i = 3 to n {
            W[i] = Max(W[i - 2] + w(vi), W[i - 1])
      }

      Create empty set S
      i = n
      While i > 0 {
            If W[i] == W[i - 1] {
                  // This means vi is not in the set, update i
```

```
                    i = i - 1
                }
                Else if W[i] == W[i - 2] + w(v_i) {
                    // This means v_i is in the set, update the set
                and i
                    Add v_i into set S
                    i = i - 2
                }

                // Do some boundary checking here for i == 1 and i ==
                0
            }

            Return S
        }
```

The runtime of this algorithm is O(n)

4. Let $G_1 = (V, E_1)$ and $G = (V, E_2)$ be two undirected graphs such that $V = \{1, 2, \cdots, n\}$. We say that G_1 is isomorphic to G_2 if there is a permutation $\Pi : \{1, 2, \cdots n\} \rightarrow \{1, 2, \cdots, n\}$ such that for every $1 \leq i \leq j \leq n$, if $\langle i, j\rangle \in E_1$, then $\langle \Pi(i), \Pi(j)\rangle \in E_2$. Show that the decision problem of whether two input graphs are isomorphic or not belongs to NP.

Ans:

Witness: A permutation Π

Verification:

- Verify Π is a permutation -> $O(n^2)$
- ∀ <x, y> ∈ E_1, check if <Π(x), Π(y)> ∈ E_2 -> $O(m^2)$
- ∀ <p, q> ∈ E_2, check if <Π^{-1}(p), Π^{-1}(q)> ∈ E_1 -> $O(m^2)$

Overall this takes $O(m^2 + n^2)$ which is polynomial.

Thus, deciding whether two input graphs are isomorphic or not belongs to NP.

~~(The reason why it's in NP and not P is because there are actually n! Permutations, which is way larger than polynomial)~~

-- **Note**: the generally believed conjecture is that Graph Isomorphism is in P (i.e., has a polynomial time algorithm). We don't have a polynomial time algorithm for this yet, unfortunately :(.

Um

5. Given an undirected graph $G = (V, E)$, a set $S \subseteq V$ is a *dominating set* if every vertex $v \in V$ either belongs to S or is adjacent to a vertex in S. We are interested in the following decision problem: Given G and k, is there a dominating set of size $\leq k$ in G. Show that this decision problem is in NP.

Ans:

Witness: A set S

Verification:

- Verify size of S is less than or equal to k -> O(1)
- Verify all vertices in S are in G -> O(kn)
- ∀ v ∈ V, check if v ∈ S -> O(mn) (worst case)
    - If v ∉ S, then ∀ <v, u> ∈ E, check if u ∈ S

Overall this takes O(nm) time, which is polynomial.

~~The reason why this is in NP rather than in P is because finding the number of sets S with size less than or equal to k is k! Which is way larger than polynomial, especially for large values of k.~~

Note: NP does not mean “not in P” (it could be that P = NP), and the fact that the number of solutions is large does not mean that the problem is not in P.

6. Suppose you are a faculty member at a large midwestern university. A student comes to your office making the following claim: "I came up with an algorithm that, given a weighted directed graph G and an integer k, produces (in polynomial time) a bipartite graph G' with the property that there is a TSP tour of total cost $\leq k$ in G if and only if there is a perfect matching in G'. " What should you conclude?

Ans:
The student is claiming to have proof that TSP reduces to Bipartite Perfect Matching. Bipartite Perfect Matching is in P (has a polynomial time algorithm). Since TSP is NP complete, the student is showing that P = NP. So it is overwhelmingly likely that the student has a flaw in their proof.

(Or you just found a way to make $1,000,000)

7. Suppose that the Bulletproof Rabbit problem has been shown to be in NP, and that Bulletproof Rabbit $\leq_p$ 3-SAT. Can we conclude that Bulletproof Rabbit is NP-complete?

Ans:
=
~~=====Yes. We have the theorem:~~
~~If A is NP complete and B reduces to A, then B is NP complete.~~
~~3 Sat is NP complete, and Bulletproof Rabbit reduces to 3 Sat, so Bulletproof Rabbit is NP complete.~~

This is **not** correct. The theorem is:
If
A is NP complete,
B is in NP, and
A reduces to B,

then B is NP complete.

In particular, you cannot conclude anything about Bulletproof Rabbit, except that it is in NP. Everything in NP reduces to 3-SAT, so it really isn't giving you any more information than that.

8. Suppose you have just proved that Vertex Cover reduces to the Invisible Canteloupe problem. Identify which additional facts you can conclude, and explain briefly.

   - Invisible Canteloupe is also reducible to Vertex Cover
   - Invisible Canteloupe is not reducible to Vertex Cover
   - There isn't enough information to tell whether Invisible Canteloupe is reducible to Vertex Cover
   - Hamiltonian Cycle is reducible to Invisible Canteloupe

Ans:

Both the third and the fourth bullet point are true.

The third bullet point is true because we do not know if Invisible Cantaloupe is in NP or not (if it is, then 1 is true and 2 is false).

The fourth bullet point is true because if A ≤ pB and B ≤ pC, then A ≤ pC.
Since Hamiltonian reduces to Vertex cover and Vertex cover reduces to Invisible Cantaloupe, thus Hamiltonian is reducible to Invisible Cantaloupe.

Question:
Why wouldn't bullet one be true0? Since vertex cover is NP-Complete, and it reduces to Invisible Cantaloupe and everything in NP reduces to vertex cover, so everything in NP reduces to Invisible Cantaloupe. Then Invisible Cantaloupe is NP Complete, so Invisible Cantaloupe can be reduced to all other NP-complete problems, i.e. it can be reduced to vertex cover? Regardless of whether or not P = NP is true, this should still hold?

- ~~Answer: We technically don't know if Invisible Cantaloupe is in NP or if it is in P. If it is in P, then it shouldn't reduce to all the other NP-Complete algorithms such as vertex cover because then it wouldn't be solved (because if A ≤ pB and A is not in P then B is not in P). This is the reason why bullet 3 is true.~~
- If anyone else was confused like I was: since VC reduces to IC, IC is NP-hard. However, because a problem is NP-hard, does not mean that it is in NP. For a problem to be NP complete, it needs to both be NP-hard and in NP. An example of this is the halting problem. It is NP-hard, but isn't in NP.

Everything in NP reduces to NP complete problems. This includes problems in P, since P is a subset of NP. So the only way for (1) to be true is if Invisible Canteloupe is in NP.

The only way for (2) to be true is if Invisible Canteloupe is **not** in NP (and therefore not in P, either).

9. Consider the following problem: There are teams of middle-schoolers coming to a big Lego construction extravanza. The site hosting the event has k work areas and a collection U of Lego bricks. Each team has a plan for what they are going to build, and a list of the bricks they need from U. Naturally, a given brick can't be used simultaneously by two different teams. You'd like to find a collection of k teams that can all work at the same time during the first shift of the event. Can this problem be solved in polynomial time? Justify your answer.

Ans:

You can say that this is the Set Packing problem, which is NP complete, and therefore is unlikely to have a polynomial time algorithm (See post 583 on Piazza).
The real answer: No, because middle schoolers haven't learned polynomials yet.

(Potential Answer?) Define each team as a node in a graph. An edge between teams (nodes) in the graph means that they need to use the same lego piece at the same time. Now we need to find k independent sets. This isn't solvable in polynomial time. **Why?** For every subset of size k, check if it is an independent set.

Witness: A set S
Verifier:

- Check if the set S is size K O(1)
- For each vertex in the set, make sure two any two vertices are not connected by an edge O(m+n) time?

This runs in polynomial time, so the problem is NP. The verification can be done in polynomial time but finding the actual subset can not.

^^^ Don't need the proof that it is NP - just the proof that it is NP-hard, I think. A P problem can also be verified in polynomial time, so proving it is in NP doesn't help. I.e. P is a subset of NP.

This looks like a similar problem: https://www.site.uottawa.ca/~lucia/courses/4105-02/a2.pdf

10. Consider the graph G in Figure 1 with edge capacities as indicated.

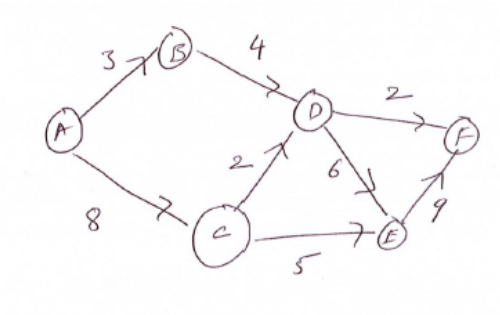


Figure 1: Graph with Flow Capacities

Suppose that we pushed a flow of 4 units in this graph as in Figure 2, where the numbers represent flow values along each edge.

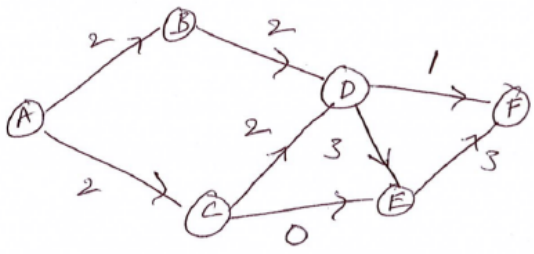


Figure 2: Graph with Flow Values

(a) Draw the residual graph

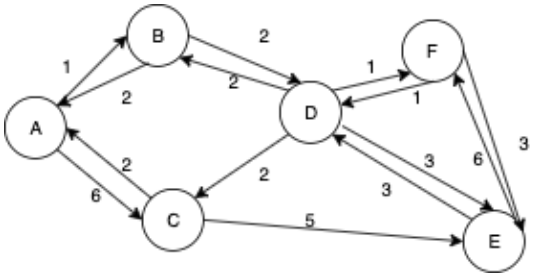
(b) Find an augmenting path P and in the residual graph

(c) Draw the flow graph obtained by augmenting the flow along P.

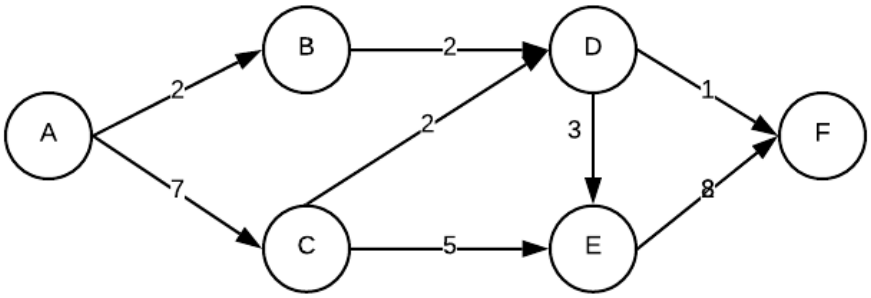
~~a. A -> B: 1~~
~~A -> C: 6~~
~~B -> D: 2~~
~~C -> D: 0~~
~~C -> E: 5~~
~~D -> E: 3~~
~~D -> F: 1~~
~~E -> F: 6~~
~~b. A -> C -> E -> F~~
~~c. Using the path above (Bold = +4)...~~
~~A -> B: 2~~
~~**A -> C: 6**~~

~~B -> D: 2~~
~~C -> D: 2~~
~~**C -> E: 4**~~
~~D -> E: 3~~
~~D -> F: 1~~
~~**E -> F: 7**~~

a) Forward edge weights indicate how much we can increase the flow through that edge. Backward edges indicate how much flow can be pushed back according to the current network. This diagram might be wrong so please verify!
--This seems right, I believe the previous answer is wrong. Residual graphs need to show both how much more flow can be pushed between each node and how much can be pushed backwards.

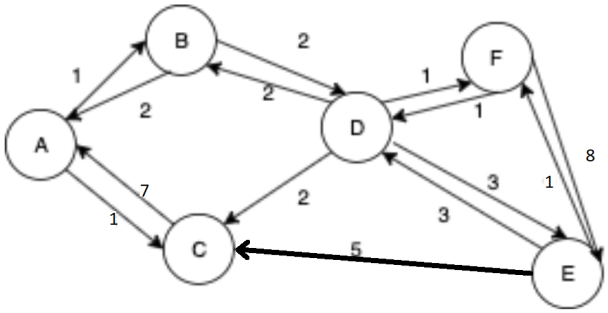


b) A -> C -> E -> F is one possible path (typically best to pick path with the largest flow increase IMO, which would be the described path)
c) new FLOW graph



If you were to redraw the residual graph after augmenting the path in B, below is what it would look like, but this is just additional info.

Bottleneck = 5 (from C to E)
New residual graph:


A
B
C
D
E
F
1
2
2
2
1
1
7
1
2
3
3
1
8
5

Required Knowledge

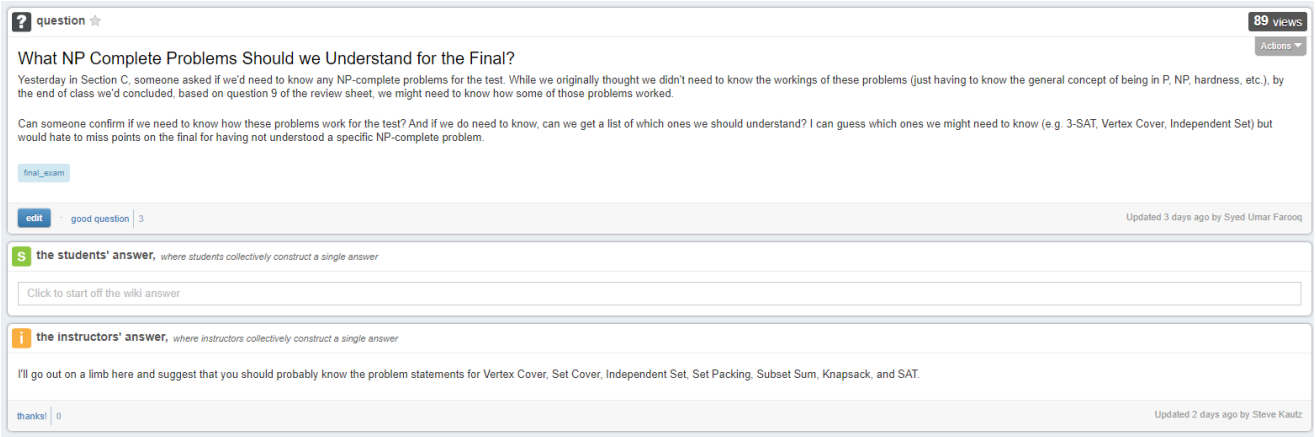
(feel free to add to this ^^)'

From the syllabus:

Course Topics (Tentative)

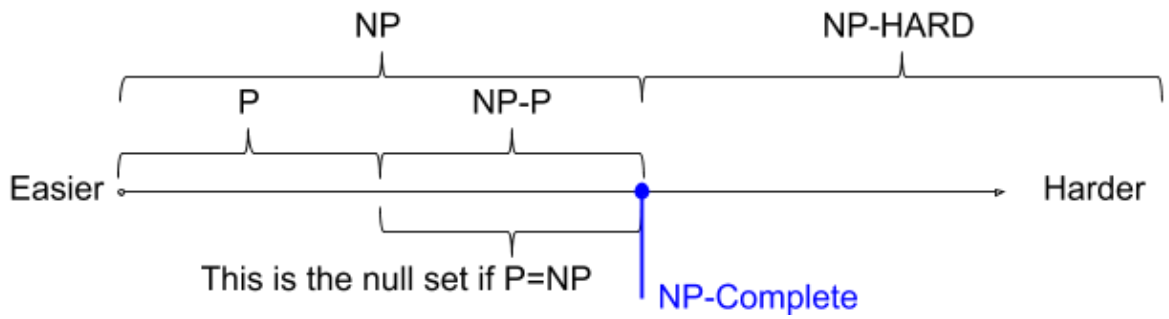
- Run time, Big O, and asymptotic bounds
- Sorting and searching
- Divide and Conquer
- Graphs and graph algorithms
- Greedy algorithms
- Dynamic programming
- NP completeness
- Approximation algorithms and Heuristics

NP Questions:



Based off of Steve's suggestions of knowing the problem statements for the following problems.

A problem Y is NP-hard if $X \leq_p Y$ for every $X \in NP$
A problem Y is NP-complete if it is NP-hard and $Y \in NP$



Made a diagram I saw in an online lecture ^^^

Vertex Cover

(covering problem) *See note on packing and covering problems on page 456 in the text.*

"Given a graph G = (V, E), we say that a set of nodes S⊆V is a *vertex cover* if every edge e ∈ E has at least one end in S.

Note that the following fact about this use of terminology: In a vertex cover, the vertices do the "covering," and the edges are the objects being "covered."

Now, it is easy to find1 large vertex covers in a graph (for example, the full vertex set is one); the hard part is to find small ones."

- Algorithm Design, Jon Kleinberg and Eva Tardos, p455
- https://www.geeksforgeeks.org/vertex-cover-problem-set-1-introduction-approximate-algorithm-2/
    - "A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either 'u' or 'v' is in vertex cover"
- Vertex Cover $\leq_p$ Independent Set

Set Cover

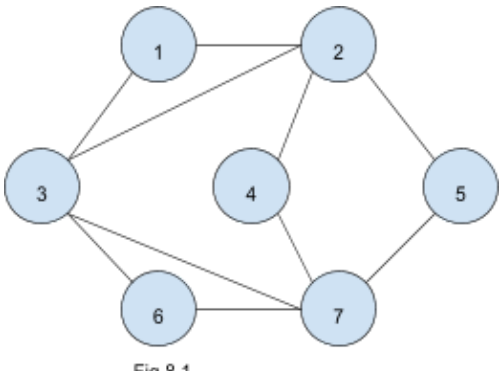
"We can phrase Set Cover as follows:

> *Given a set U of n elements, a collection $S_1,\ldots, S_m$ of subsets of U, and a number k, does there exist a collection of at most k of these sets whose union is equal to all of U?*

Imagine, for example, that we have *m* available pieces of software, and a set U of *n capabilities* that we would like our system to have. The i^{th} piece of software includes the set $S_i \subseteq U$ of capabilities. In the Set Cover problem, we seek to include a small number of these pieces of software on our system, with the property that our system will then have all *n* capabilities."

- Algorithm Design, Jon Kleinberg and Eva Tardos, p455
- Vertex Cover $\leq_p$ Set Cover

  See text p456-457 for more details. [There's visualisation of the set cover problem that is a hassle to draw in Google insert drawing lol, maybe i'll take a pic later or smth]



Independent Set

(packing problem)

“Recall [from chapter 1] that in a graph G = (V, E), we say a set of nodes S⊆V is *independent* if no two nodes in S are joined by an edge. It is easy to find small independent sets in a graph (for example, a single node forms an independent set); the hard part is to find a large independent set, since you need to build up a large collection of nodes without ever including two neighbours. For example, the set of nodes {3.4.5} is an independent set of size 3 in the graph in Figure 8.1, while the set of nodes {1,4,5,6} is a larger independent set.”

- Algorithm Design, Jon Kleinberg and Eva Tardos, p454
- Independent Set $\leq_p$ Vertex Cover

Set Packing

“Just as Set Cover is a natural generalisation of Vertex Cover, there is a natural generalisation of Independent Set as a packing problem for arbitrary sets. Specifically, we define the *Set Packing* problem as follows:

> *Given a set U of n elements, a collection S_1,..., S_m of subsets of U, and a number k, does there exist a collection of at least k of these sets with the property that no two of them intersect?*

In other words, we wish to “pack” a large number of sets together, with the constraint that no two of them are overlapping.

As an example of where this type of issue might arise, imagine that we have a set U of *n* non-sharable *resources*, and a set of *m* software processes. The i^{th} process requires the set $S_i \subseteq$ U of resources in order to run. Then the Set Packing Problem seeks a large collection of these processes that can be run simultaneously, with the property that none of their resource requirements overlap (i.e., represent a conflict). “

- Algorithm Design, Jon Kleinberg and Eva Tardos, p458-459
- Independent Set $\leq_p$ Set Packing

Subset Sum

(special case of Knapsack)

“We can formulate a decision version of this problem [knapsack] as follows:
Given natural numbers w_1, … , w_n, and a target number W, is there a subset of {w_1, … , w_n} that adds up to precisely W?”

- Algorithm Design, Jon Kleinberg and Eva Tardos, p491
- Please read p491-493 for more information on the subset sum problem and a proof of why it is NP-Complete.

Knapsack

6.4 Dynamic Programming

“Consider a situation in which each item *i* has a nonnegative weight w_i as before, and also a distinct *value* v_i. Our goal is now to find a subset S of maximum value $\sum_{i \in S} v_i$, subject to the restriction that the total weight of the set should not exceed W: $\sum_{i \in S} w_i \leq W$.

- Algorithm Design, Jon Kleinberg and Eva Tardos, p271-272

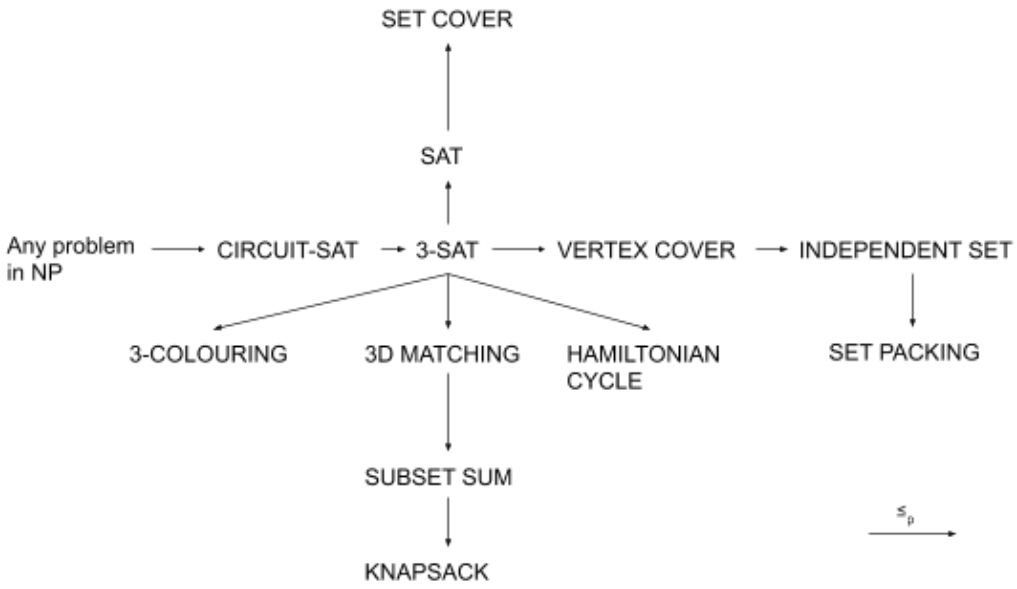
SAT
(also referred to as the Satisfiability Problem)
"Given a set of clauses $C_1, \dots, C_K$, over a set of variables $X = \{x_1, \dots, x_n\}$, does there exist a satisfying truth assignment?"

3-SAT: a case in which all clauses contain exactly three terms.
"Given a set of clauses $C_1, \dots, C_K$, each of length 3, over a set of variables $X = \{x_1, \dots, x_n\}$, does there exist a satisfying truth assignment?"

- Algorithm Design, Jon Kleinberg and Eva Tardos, p459-460
- 3-SAT $\leq_p$ Independent Set

NP-Complete Flow Diagram

A flow diagram from Steve's lectures:



IMPORTANT: all the problems listed above (Circuit-Sat, 3-colouring,....Independent Set, Set Packing, Set Cover, etc.) are NP-complete. Therefore each problem reduces to every other problem (they are all, in a sense, equivalent).

How to get 100% on the exam