# CSCI 270 Final Exam Cheatsheet

March 22, 2023

## Pre-MT Materials

### 1. Stable Matching

- A matching is **stable** if for each pair $(u, s)$ not assigned, either $u$ prefers its assigned node or $s$ or $s$ prefers its assigned node over $u$.
- Stable matching is *not* unique: if $A$ prefers 1, $B$ prefers 2, 1 prefers $B$, and 2 prefers $A$, then $\{(A,1),(B,2)\}$ and $(A,2),(B,1)$ are both stable.
- **Gale-Shapley**: (1) pick any single man $m$, (2) proposed to highest ranked women $w$ not yet proposed, (3) if $w$ is single OR prefers $m$ over her current partner, match $(m, w)$, (4) repeat until done.
- Properties of G-S: terminates, stable, & men-optimal.

### 2. Greedy Algorithms

- "Pick whatever seems best at the moment."
- "Greedy stays ahead" (proof technique): in each stage greedy performs no worse than optimal solution.
- **Interval selection**: given $[a_i, b_i]$, pick as many as possible without overlap. Greedy solution: sort by finish times from earliest to latest, and pick the interval as long as it does not conflict with earlier ones.
- **MST construction**: construct MST of $G = (V, E)$.
  - **Kruskal**: (i) sort edges by increasing cost, and (ii) iteratively pick the cheapest as long as it does not create a cycle.
  - **Prim**: start with any $S = \{s\}$; iteratively find and add the cheapest edge in cut $(S, S')$.

### 3. Divide & Conquer

- "Break questions into smaller ones until they are small enough to be solved directly."
- Most familiar example: merge sort.

**Theorem: Master Theorem**

Let $a \geqslant 1, b > 1$. Assume some recursion relation complexity satisfies

$$T(n) = aT(n/b) + f(n), T(1) = \Theta(1).$$

(MT1) If $f(n) = \mathcal{O}(n^{\log_b(a-\epsilon)})$ for some $\epsilon > 0$ then $\boxed{T(n) = \Theta(n^{\log_b a})}$.

(MT2) If $f(n) = \Theta(n^{\log_b a})$ then $\boxed{T(n) = \Theta(n^{\log_b a} \log n)}$.

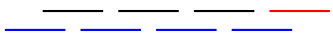
(MT3) If $f(n) = \Omega(n^{\log_b(a+\epsilon)})$ for some $\epsilon > 0$ with $\limsup_{n\to\infty} \frac{af(n/b)}{f(n)} < \infty$, then $\boxed{T(n) = \Theta(f(n))}$.

- Intuition: (MT1) says $f(n)$ is overwhelmed by $n^{\log_b a}$ small tasks like $T(1)$; (MT2) says $f(n)$ and small tasks have similar workload; (MT3) says most work is done by $f(n)$.
- **Closest pair of points**: given $n$ points on $\mathbb{R}^2$, find a pair with closest Euclidean distance?

  Solution in a nutshell: divide points to left and right, and do some math in the middle.

### 4. Dynamic Programming

- "In order to achieve optimal solution at this step, what must be done by the previous step?"
- Memoization: storing values for future recursive calls.
- **Interval election with weights** $w_j$:
  - Greedy doesnt work — if all non-red intervals have weight 2, the optimal solution depends on the weight of the red one.



  - Observation: (i) if OPT includes interval $i$, then it does not include any intersecting $i$; if OPT does not, pretend as if interval $i$ never existed.
  - $\text{OPT}(j) = \max\{w_j + \text{OPT}(p(j)), \text{OPT}(j-1)\}$ where $p(j)$ is the largest $i < j$ where intervals $i, j$ are disjoint. Using memoization this can be done with $\mathcal{O}(n)$ time and space.
- **Knapsack**: given items with weights $w_j$ and values $v_i$, maximize the total items taken subject to a weight constraint $W$. For each integer $0 \leqslant w \leqslant W$ and $1 \leqslant j \leqslant n$ consider the subproblem $\text{OPT}(i, w)$:

  $\text{OPT}(i, w) = \max\{\text{OPT}(i-1, w), v_i + \text{OPT}(i-1, w-w_i)\}$.

  Since we have two variables, we implement this using a **tabular** method, keeping track of a $n \times W$ array.
- **String alignment** (sketch): three possibilities for $\text{OPT}(i, j)$, the optimal cost of aligning first $i$ letters of word $x$ with first $j$ letters of word $j$:
  - $x[i]$ aligns with $y[j]$: no extra cost compared to $\text{OPT}(i-1, j-1)$.
  - $x[i]$ aligns with a blank: extra cost of removing a letter from $x$ compared to $\text{OPT}(i-1, j)$.
  - $y[j]$ aligns with a blank: extra cost of adding a letter to $x$ compared to $\text{OPT}(i, j-1)$.
  - Take minimum, and set up base cases...

### 5. Max-Flow, Min-Cut, & Duality

**Definition: Flow (on graph)**

Given a graph $G = (V, E)$, an $s-t$ **flow** is a function $f : E \to \mathbb{R}$ satisfying

- (conservation) for all nodes $v \neq s, t$,

$$\sum_{e \text{ out of } v} f(e) = \sum_{e \text{ into } v} f(e), \text{ and}$$

- (constraints) $0 \leqslant f(e) \leqslant c(e)$ for all edges $e$, where $c(e)$ is the capacity of $e$.

We define the **value** of a flow, $\nu(f)$, to be the total flow out of the source (or into the sink).

**Definition: Cut (on graph)**

An $s-t$ **cut** $(S, S^c)$ is a partition of $V$ with $s \in S$ and $t \in S^c$. An edge $e = (u, v)$ **crosses** the cut if $u \in S, v \in S^c$. The total **capacity** of the cut is defined by $c(S, S^c) := \sum_{e \text{ crosses } (S,S^c)} c(e)$.

**Theorem: Max-Flow / MinCut (Ford-Fulkerson)**

Given a graph $G$ with nonnegative edge capacities and nodes $s, t$:

- max $s-t$ flow and min $s-t$ cut can be found in polynomial times,
- if $c(e)$ are integers, then the algorithm outputs an integer max-flow, and
- max-flow value equals min-cut capacity.

**Algorithm 1:** Ford-Fulkerson Algorithm

```
Start with zero flow, i.e. f(e) = 0 for all edge.
while residual graph G(f) contains an s – t path do
    Let P be one such path in G(f) [BFS/DFS]
    Augment f along P
Function residual_graph(G = (V, E)):
    Start with empty G(f)
    for each edge e = (u, v) do
        if f(e) > 0: add (v, u) to G(f) with capacity f(e).
        if f(e) < c(e): add (u, v) to G(f) with capacity c(e) – f(e).
Function augment(f, P):
    Find ẽ = argmin_{e∈P} f(e)
    if ẽ is a forward edge: increase f on P by f(ẽ).
    if ẽ is backwards: decrease f on P by f(ẽ).
```

- **Duality**: since any flow and any cut on $G$ satisfies $\nu(f) \leqslant c(S, S^c)$ , we have $\max \nu(f) \leqslant \min c(S, S^c)$ so if equality can be attained, we must have simultaneously found max-flow and min-cut.
- (Applcation) **maximum bipartite matching problem**: to find a maximum bipartite matching between sets $X$ and $Y$, we create a source $s$ and sink $t$. We connect $s$ to each node in $X$ with capacity 1, $v$ to each node in $Y$ with capacity 1, and fully connect nodes between $X$ and $Y$ with infinite capacity. Now find a max-flow.
- [See below] Problem with Ford-Fulkerson: it is pseudopolynomial — if the capacity of edges reach $10^{100}$ it may need to iterate $10^{100}$ times!
- Workaround: always choose the widest path (weakly polynomial) or always choose the shortest path (known as **Edmonds-Karp** algorithm, runs in strongly polynomial, $\mathcal{O}(m^2 n)$ ).
- **Pseudopolynomial** runtime example: DP Knapsack with $W = 2^{100}$ , where $W$ can be stored in 100 bits, but running it takes ... (recall $\mathcal{O}(nW)$ ). Still, this *is* polynomial in $n, W$. [Weakly/strongly polynomial not included, as they were barely mentioned in lectures.]

### Application: Image Segmentation

- Question: given an image (many pixels), is each particular pixel foreground or background?

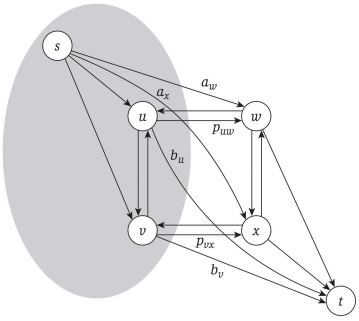
- Formulation: given $G = (V, E)$, each node $v$ has a foreground score $a(v) \geqslant 0$ that we get from labelling $v$ with "A", a background score $b(v) \geqslant 0$ for labelling it with "B". For each edge $e = (u, v)$, penalty $p(e) \geqslant 0$ for labelling $u, v$ with different labels. Find a partition $(S, S^c)$ maximizing

$$Q(S, S^c) = \sum_{v \in S} a(v) + \sum_{v \notin S} b(v) - \sum_{\substack{e=(u,v) \\ e \in S, v \notin S}} p(e).$$

- Solution: since

$$Q(S, S^c) = \sum_v a(v) + \sum_v b(v) - \left( \sum_{v \notin S} a(v) + \sum_{v \in S} b(v) + \sum_{\substack{e=(u,v) \\ v \in S, u \notin S}} p(e) \right)$$

it suffices to minimize the red terms, which we call $R(S, S^c)$. But this is just a min-cut: (i) add source $s$, sink $t$, (ii) connect $s$ with each $v$ with capacity $a(v)$, (iii) connect all $v$ to $t$ with capacity $b(v)$, (iv) find a min $s-t$ cut $(\{s\} \cup S, \{t\} \cup S^c)$, and (v) return $S$. See the image from KT's book.




## 6. NP-Completeness

- We focus on **decision problems**, i.e., those outputting YES or NO.
- Decision problems are polynomial-time equivalent to optimization ones (compute the optimum): do binary search for the right value.
- **P**: problems solvable in polynomial time. **NP** stands for **nondeterministic polynomial time** (Kempe said don't tell anyone you studied with him if you still say NP is "not polynomial").
- Another definition for NP: problems with **efficient certifiers**, i.e., "a proposed solution (**certificate**) can be verified in polynomial time." Formal definition:

**Definition: Efficient Certifier**

An efficient certifier for a problem $X$ is a polynomial-time algorithm $B(x, y)$ with inputs $x$ (regular input for $X$) and $y$ (certificate) such that

(1) if the correct answer for $x$ is YES, then there exists one $y$ with $|y| \leqslant r(|x|)$ such that $B(x, y)$ answers YES;

(2) if the correct answer for $x$ is NO, then for every $y$, $B(x, y)$ answers NO; and

(3) $r$ is polynomial.

- Efficient certifier example:

  $X$: "given graph $G$ and target cost $C$, is there a spanning tree with cost $\leqslant C$?"

  Certificate: $y$, a proposed spanning tree.

  Verification process: check (i) $y$ is a spanning true, and (ii) cost of $y \leqslant C$. If both are satisfied, return YES, else return NO.

  Efficient? Show this verification process runs in polynomial time.

**Proposition: P $\subset$ NP**

*Proof. Let $X \in P$ and let $A$ be an algorithm that solves $X$ in polynomial time. Let $B$ be the trivial certifier such that $B(x, y)$ copies $A(x)$'s answer.*

- *Clearly efficient.*
- *If $x$ is a YES answer for $X$, then $B(x, y)$ answers YES (or any, so at least one $y$).*
- *If $x$ is a NO answer for $X$, then $B(x, y)$ answers NO for all $y$.* □

**Definition: Polynomial-Time Reduction (Karp)**

We say $X$ is **polynomial-time reducible** to $Y$, written $X \leqslant_p Y$, if we can solve $X$ by calling a "black box" solver of $Y$ a polynomial number of times.

*Prof. Kempe's equivalent definition [which I find to be slightly more elusive]: a Karp reduction from $X$ to $Y$ is a polynomial $f$ with the following properties:*

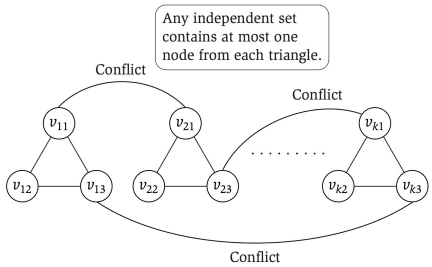
- *The input of $f$ is an input to $X$; the output of $X$ is an input to $Y$.*
- *If $x$ is a YES input for $X$, if and only if $f(x)$ is a YES input for $Y$.*

- $\leqslant_p$ is transitive.
- Think of $\leqslant_p$ as "level of hardness": bigger means "harder."
- A problem is **NP-complete** if (i) it is NP and (ii) for all $X \in$ NP, $X \leqslant_p$ this problem. A problem is **NP**-hard if it satisfies (ii) [but not necessarily (i)].

### NP-Complete Problems

- To show $Y$ is NP-complete:
  - Show $Y$ is NP by giving an efficient certifier.
  - Unless you are crazy, pick a known NP-complete problem $X$, and reduce from $X$ to $Y$, i.e., showing $X \leqslant_p Y$.
- Cook-Levin: **SAT** and **3SAT** are NP-complete.
- 3SAT example: is $(x_1 \vee \overline{x}_2 \vee \overline{x}_4) \wedge (x_2 \vee x_3 \vee x_4)$ satisfiable? [$\vee$ as "or", $\wedge$ as "and"]
- The following are NP-complete:
  - **INDEPENDENT SET**: "given $G$ and $k \in \mathbb{N}$, does there exist $k$ nodes such that no two nodes are connected by an edge?"

    3SAT $\leqslant_p$ INDEPENDENT SET: [image from book] conflict refers to negations of literals, e.g. $x_4$ and $\overline{x}_4$ from the 3SAT example above.




  - **VERTEX COVER**: "given $G$ and $k \in \mathbb{N}$, does there exist $k$ nodes such that each edge has at least one endpoint among these nodes?"

    INDEPENDENT SET $\leqslant_p$ VERTEX COVER: ($S$ is an independent set) iff (each edge has $\geqslant 1$ endpoint in $S^c$) iff ($S^c$ is a vertex cover), so given INDEPENDENT SET with $G, k$, transform to $G', \{S_v\}, n - k$, with $S_i$ the set of nodes adjacent to $v$.
  - **SET COVER**: given a set of $U$ elements $S_1, \ldots, S_n \subset U$, and $k \in \mathbb{N}$, does there exist $k$ sets whose union cover $U$?

    VERTEX COVER $\leqslant_p$ SET COVER: given inputs $G, k$ to VERTEX COVER, transform into $V, \{S_v\}, k$ like above.

## 7. Undecidability

- A (yes/no) function is **computable** if there exists a program that tells if each input leads to YES or NO.
- Uncountable functions, countable programs (finite strings) $\Rightarrow$ "most" functions are not computable.
- **HALT** (diagonal halting): given a program $P$, does $P(P)$ terminate?

  HALT is undecidable. Suppose for contradiction that a program SOLVER solves HALT. Define

```
1  int Destroyer (string P) // program P {
2      if (Solver says P terminates):
3        while (true); // infinite loop
4      else return 0;
5  }
```

  If DESTROYER(DESTROYER) terminates then the program is stuck at line 3 and never terminates; if DESTROYER(DESTROYER) doesn't, the program terminates at line 4. Contradiction!

- **Many-to-one reduction**: $X \leqslant_m Y$ if there exists some $f$ mapping inputs to $X$ to inputs to $Y$ such that (i) YES instances map to YES instances, (ii) NO to NO, and (iii) $f$ always terminates. (Similar to Karp!)
- **TOTAL**: "does a program $P$ terminates for all inputs?"
  - TOTAL is undecidable: HALT $\leqslant_m$ TOTAL.
  - Reduction:

```
1  int Q (string x) {
2      Run P(P) // doesn't even look at x
3      return 0;
4  }
```

  - If $P(P)$ terminates, then $Q(x)$ terminates and returns 0 for all $x$, i.e., $Q$ is TOTAL.
  - If $P(P)$ does not, then $Q(x)$ does not terminate for any (in particular some) string, so $Q$ is not TOTAL.

**GOOD LUCK!**