Second Midterm Solutions

Name:

SID:

GSI and section time:

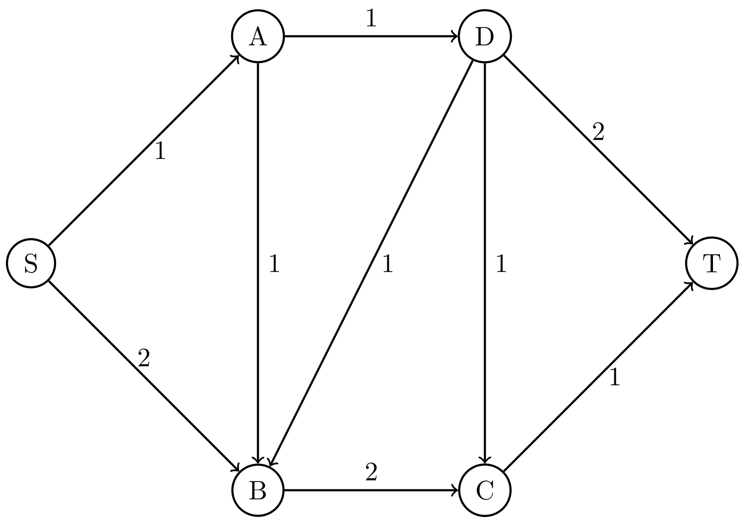
Read the questions carefully first. Be precise and concise. The number of points indicate the amount of time (in minutes) each problem is worth spending. Not all parts of a problem are weighted equally. Write in the space provided, and use the back of the page for scratch. Box numerical final answers.

Some questions are marked as “extra-credit”. Only attempt these questions at the very end.

Good luck!

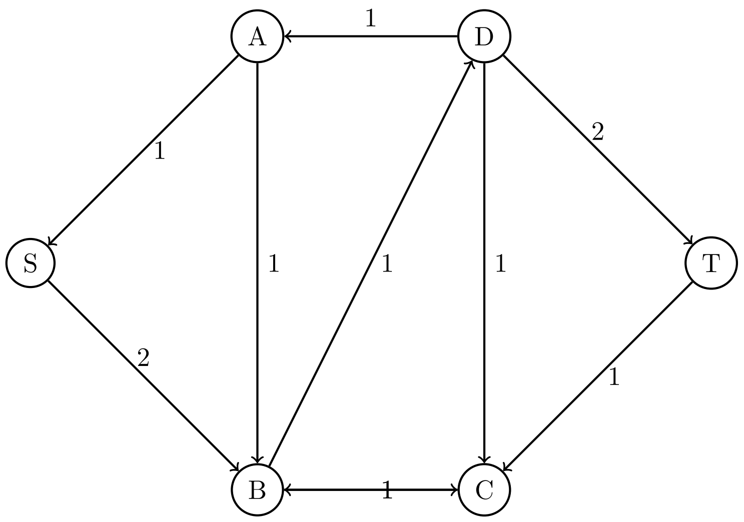
When an explanation is required, answer with one or two short sentences. *(25 points)*

1. Consider the following directed graph ~~with all edge capacities equal to 1~~.



In the first step of Maximum-Flow algorithm, we increase the flow along $S \to A \to D \to B \to C \to T$ by one unit.

(a) Draw the residual graph after this step.



(b) What happens next in the execution of Max-Flow algorithm? (the algorithm does not necessarily run for three steps)

- Send 1 unit of flow on path $S \to B \to D \to T$
- Send unit(s) of flow on path $S \to$ $\to T$
- Send unit(s) of flow on path $S \to$ $\to T$

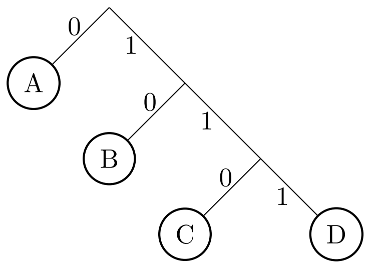
(c) What is the minimum $S - T$ cut in the graph?

Solution: There were two correct answers:

- S-side of the partition = $\{S, A, B, C\}$
- T-side of the partition = $\{T, D\}$
- S-side of the partition = $\{S, B, C\}$
- T-side of the partition = $\{T, A, D\}$

2. Give a set of frequencies on the four symbols $\{A, B, C, D\}$, for which the Huffman coding tree would be as shown below.

Symbol	Frequency
A	0.5
B	0.25
C	0.125
D	0.125



3. On a weirdly designed keyboard, every insertion takes 2 keystrokes, every deletion takes 3 key strokes and every substitution takes 4 keystrokes. Write the recurrence relation for the edit distance (minimum number of key strokes needed to edit a string $x[1, \ldots, m]$ in to a string $y[1, \ldots, n]$).

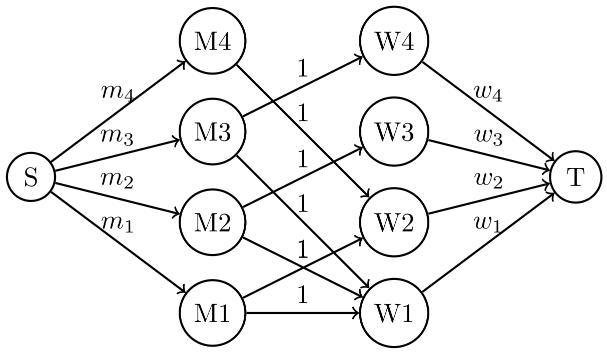
Solution: $ED[i,j] = \min \begin{cases} ED[i-1, j] + 3 \\ ED[i, j-1] + 2 \\ ED[i-1, j-1] + 4 \cdot \text{diff}(i, j) \end{cases}$

Dating with Flows (5 points + 5 extra credit points)

4. A dating website has used complicated algorithms to determine the compatibility between the profiles of men and women on their website. The following graph shows the set of compatible pairs.

   The website is trying to setup meetings between the men and the women. The i^{th} man has indicated a preference of meeting exactly m_i women, while the j^{th} woman prefers to meet at most w_j men. All meetings must be between compatible pairs.

   (a) How would you use the Max-flow algorithm to set up the meetings? (Draw the graph on which you would run the Max-flow) Briefly justify your answer.



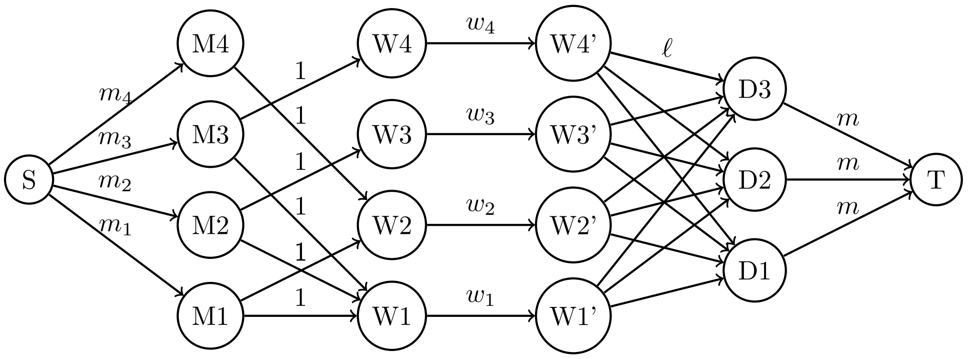
Solution: We design a flow network, shown above, such that each unit of flow indicates a meeting between a man and a women. Observe that:

- Edges (S, M_i) have capacity m_i, enabling man i to be matched with up to m_i women.
- Since each man-woman pair should only be matched at most once, edges (M_i, W_j) have unit capacity.
- Edges (W_j, T) have capacity w_j, preventing woman j from being matched with more than w_j men.

We run a max flow algorithm. If the flow f equals $\sum_i m_i$, whether there is flow on edge (M_i, W_j) indicates whether that meeting takes place in the schedule. This is well-defined because the integral capacities imply an integral max flow.

Importantly, why does maximizing *global* flow also satisfy each *individual* man's demand to meet with exactly m_i women? Because the edge capacities on (S, M_i) prevent any man from taking on more dates than he requires; thus if there is enough flow to accommodate all men, such a flow will be the global max flow.

(b) *(Extra Credit, Attempt at the end, 5 points)* Suppose all the meetings need to be scheduled over 3 days and no woman wants to meet more than ℓ men on the same day. Moreover, the dating website cannot host more than m meetings in total on the same day. How would you compute the schedule using Maxflow? (Assume that the men don't mind meeting any number of women on the same day)



Solution: As in part (a), we design a flow network that models the constraints.

- Edges (S, M_i) with capacity m_i serve the same purpose as in part (a).
- Unit-capacity edges between men and women are also the same as in part (a).
- Each woman W_j now has an edge to a copy W'_j, with capacity w_j. This serves the same purpose as in part (a), though now they are not routed immediately to T.
- Nodes D_k model the three days. Each woman, in addition to her personal constraint w_j, also never wants more than ℓ dates in a single day, hence capacity ℓ on every edge (W'_j, D_k).
- Finally, each day can only take on m dates, hence the final edges to T.

As in part (a), we can run max flow and check the assignment of flow to the man-woman edges. This time, to fully specify a schedule across the three days, we also have to look at the edges (W'_j, D_k) to see how many dates each woman should have on each day.

Covering with unit intervals (10 points)

5. Given n real numbers $x_1 < x_2 < \ldots < x_n$, we would like to cover them with the minimum number of intervals of length 1.

   For example, given $\{0.1, 0.8, 4.3, 5.1, 7.1, 7.6, 8.1\}$ we can cover it with three intervals $[0, 1]$, $[4.2, 5.2]$ and $[7.1, 8.1]$.

   (a) The greedy strategy used for the set cover problem can be applied to this problem. Give an example where this fails to find the minimum covering, and show the execution of the greedy strategy.
   Solution: Let the numbers be $\{1.0, 1.9, 2.0, 2.5, 2.9, 3.2\}$
   The optimal solution is to use the two intervals $[1.0, 2.0], [2.2, 3.2]$.
   The greedy strategy would first use the interval $[1.9, 2.9]$, since this would cover 4 points. However, this would leave the two points 1.0 and 3.2, which would require two additional intervals to cover.

   (b) Describe a different greedy strategy that always finds the minimum covering. (no proof necessary)

   Solution: While any points are not covered, add the interval that starts with the smallest uncovered point.

6. True or false? Circle the right answer. No explanation needed (15 points)

(No points will be subtracted for wrong answers, so guess all you want!)

1) **T** **F** The running time of a dynamic program is at most the number of edges in its underlying DAG.
Solution: F; consider a DP algorithm that, at each subproblem, examines all *pairs* of previous subproblems (so, each node v in the underlying DAG performs $(\text{indegree}(v))^2$ work). Then the running time could be $\Omega(|E|^2)$.

2) **T** **F** There are two ways to implement any DP algorithm: bottom-up, and via recursion with memoization. Both always have asymptotically the same time and space complexity.
Solution: F; a bottom-up approach can discard the results of subproblems that will never be revisisted, thus getting smaller space complexity (e.g. edit distance can be done in $O(n)$ space).

3) **T** **F** Someone gives you a flow f on a graph G, claiming that f is a maximum flow. It is possible to verify this claim in $O(|E|)$ time.
Solution: T; we check if the target is reachable from the source via edges that are not fully used.

4) **T** **F** Not all linear programs can be solved in polynomial time.
Solution: F; there exists a polynomial-time solver for all LPs.

5) **T** **F** If a linear program has unbounded feasible region, then it does not have an optimum solution of finite value.
Solution: F; Consider $\min x + y$ subject to the constraints $x, y \geq 0$

6) **T** **F** In successive iterations of the Maxflow algorithm, the total flow passing through a vertex in the graph never decreases.
Solution: F; later iterations may reroute some existing flow so that the flow that passes through a particular vertex decreases.

7) **T** **F** The running time of the algorithm for All-Pairs-Shortest-Paths would increase by a factor of 1000, if we switch the unit of measuring distances from kilometers to meters.
Solution: F; the APSP algorithm's runtime is independent of the lengths of the edges.

8) **T** **F** The running time of the algorithm for Knapsack would increase by a factor of 1000, if we switch the unit of measuring weights from kilograms to grams.
Solution: Either; the DP algorithm for Knapsack's runtime is $O(nW)$ and proportional to the capacity of our bag, but if we use memoization, we will not have to look at every subproblem.

9) **T** **F** Given a Horn-SAT instance with n variables $\{x_1, \ldots, x_n\}$ and just one constraint $(x_1 \Rightarrow x_2)$, the Horn-SAT algorithm would set x_1 and x_2 to true, and all other variables to false.
Solution: F; all variables would be set as false. (The alternative is also a valid solution, but not the one generated by the algorithm).

10) **T** **F** If $(1, 1, 1)$ and $(2, 2, 2)$ are feasible solutions to a linear program on 3 variables then $(3, 3, 3)$ is also one.
Solution: F; imagine the LP where there are three constraints of the form $x_i \leq 2$.

11) **T** **F** If $(1, 1, 1)$ and $(3, 3, 3)$ are feasible solutions to a linear program on 3 variables then $(2, 2, 2)$ is also one.
Solution: T; any point on a line segment connecting two feasible vertices is also feasible.

12) **T** **F** The residual graph of a maximum flow f can be strongly connected.
Solution: F; if there is a path from s to t, the flow is not maximal.

13) **T** **F** The dynamic programming algorithm for the Travelling Salesman problem uses exponential amount of memory.
Solution: T; there are an exponential number of subproblems.

14) **T** **F** The value of edit distance between two strings of length n can be computed using $O(n)$ memory.
Solution: T; at most two columns of the edit distance table need to be maintained at any given time.

15) **T** **F** "Dynamic programming" sounds cool.
Solution: T; however, we were lenient towards incorrect answers.

Water Supply via Linear Programming (15 points)

7. There are four major cities and three water reservoirs in California.

   - Reservoir i holds G_i gallons of water, while city j needs D_j gallons of water.
   - It costs p_{ij} dollars per gallon of water supplied from reservoir i to city j.
   - The capacity of the piping from reservoir i and city j can handle at most c_{ij} gallons.
   - In view of fairness, no city must get more than 1/3rd of all its water demand from any single reservoir.

   Write a linear program to determine how to supply the water from the reservoirs to the cities, at the lowest cost.

   (a) What are the variables of the linear program, and what do they indicate?

   Solution:
   We define f_{ij} to be the amount of water, in gallons, supplied from reservoir i to city j.
   Common Mistakes:

   - Many students included the provided parameters (e.g. G_i, p_{ij}) or the number of cities/reservoirs as variables. These are not variables of the linear program, but parameters; the linear program cannot set these variables as part of the optimization process (e.g. if p_{ij} was a variable, then the optimal solution is to set it to 0).
   - Some students defined multiple new groups of variables, some of which were redundant (e.g. in addition to defining f_{ij} as above, but also g_i as the total flow out of reservoir i). This wasn't incorrect, but this often led to errors in part c, as described below.

   (b) What is the objective function being maximized/minimized?

   Solution:
   We wish to minimize $\sum_{ij} p_{ij} \cdot f_{ij}$, the cost of supplying the water.

(c) What are the constraints of your linear program? (No need to list every constraint, but list one of each type and explain how the rest are generated)

Solution:
We have five groups of constraints. First, the constraints:

$$\forall i : \sum_j f_{ij} \leq G_i$$

$$\forall j : \sum_i f_{ij} \geq D_i$$

$$\forall i, j : f_{ij} \leq c_{ij}$$

$$\forall i, j : f_{ij} \leq D_j/3$$

$$\forall i, j : f_{ij} \geq 0$$

The first group corresponds to the limit on how much water each reservoir can support.
The second group corresponds to the requirement of how much water each city must get.
The third group corresponds to the maximum capacity of the piping.
The fourth group corresponds to the fairness requirement.
The fifth group corresponds to a sanity check on flow; we cannot send negative amounts of flow.

Common Mistakes:

- Many students did not include the fifth constraint.
- Conversely, some students imposed nonnegativity constraints on the input parameters. This does sanity check the input, but was not needed in the solution.
- Some students simply neglected one or more of the constraints, particularly the constraints for D and G.
- It is important to emphasize that, in general, linear programs cannot have strict inequalities as constraints (e.g. $>$ and $<$ cannot appear in a linear program). None of the constraints in the problem needed a strict inequality.
- If in part (a), the student defined multiple variables, it was important to explicitly relate these variables as constraints. For example, if f_{ij} and g_i were defined as in the common mistake above, then a constraint $\sum_i f_{ij} = g_i$ was needed; otherwise, the two variables would not be related in the LP.

All-Pairs Shortest Path Again (15 points)

8. Here we will design a slightly slower, but intuitively simpler algorithm for All-Pairs Shortest Path in a graph G. The input is a graph $G = (V, E)$ with edge weights $c(i, j)$ between vertices i and j.

   Define the subproblem as follows:

   $$d(i, j, \ell) = \text{length of the shortest path from } i \text{ to } j \text{ that uses } < 2^\ell \text{ intermediate nodes.}$$

   By definition, $d(i, j, 0) = c(i, j)$.

   Solution:

   (a) Length of shortest path from i to j = $d(i, j, \underline{\quad \log |V| \quad})$

   Comments:

   - Using $\log |E|$, $|E|$ or $|V|$ would work as well, but among them only $\log |E|$ and $\log |V|$ are tight enough and would give the proper running time.
   - ℓ is not part of the input, so every expression including ℓ is incorrect.

   (b) Write a recurrence relation for $d(i, j, \ell)$.

   $$d(i, j, \ell) = \min_{k \in V} \{d(i, k, \ell - 1) + d(k, j, \ell - 1)\}$$

   Comments:

   - Adding $d(i, j, \ell - 1)$ in the minimum of the relation above is correct but redudant, since $d(i, j, \ell - 1) = d(i, i, \ell - 1) + d(i, j, \ell - 1)$.
   - Another natural approach would be to take the minimum over the edges, i.e.:

     $$d(i, j, \ell) = \min_{(u,v) \in E} \{d(i, u, \ell - 1) + c(u, v) + d(v, j, \ell - 1)\}$$

     but note that this recurence is not entirely correct, since if we set $\ell = 1$ it will allow us to use $2 (\not< 2^1)$ intermediate vertices.
   - Some people, used the following recurrence:

     $$d(i, j, \ell) = \min_{k \in V} \{d(i, k, \ell - 1) + c(k, j)\}$$

     this recurrence is incorrect, because it does not account for all the paths of $< 2^l$ intermediate nodes. In the recursive call we take into consideration all paths of $< 2^{l-1}$ intermediate nodes, but then we expand those paths by only one node so overall we take into account only paths of $< 2^{l-1} + 1$ intermediate nodes.
   - A common incorrect recurrence that we came acrross was:

     $$d(i, j, \ell) = \min\{d(i, \ell, \ell - 1) + d(\ell, j, \ell - 1)\}$$

     Although this relation tries to capture the idea of one new intermediate node and $< 2 \cdot 2^{l-1}$ intermediate nodes from the recursive calls, it has a serious type error and makes no sense. The first two arguments of $d(., ., .)$ are supposed to be vertices, while the third one is supposed to be an upper bound on the log of the intermediate nodes, so as you can see the two first arguments are of different type from the third and so a recursive call using the third argument in the place of the second, could never be correct, because the types are incorrect. Another reason for which it makes no sense, is that this relation is just the minimum of one term, so actually it is just the assignement: $d(i, j, \ell) = d(i, \ell, \ell - 1) + d(\ell, j, \ell - 1)$.

(c) Write pseudocode for computing the All-Pair-Shortest-Paths algorithm using the above recurrence.

```
for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
        $d(i, j, 0) \leftarrow c_{ij}$

for $\ell = 1$ to $\log n$ do

    for $i = 1$ to $n$ do

        for $j = 1$ to $n$ do

            $d(i, j, \ell) \leftarrow +\infty$

            for $k = 1$ to $n$ do

                $d(i, j, \ell) \leftarrow \min(d(i, j, \ell), d(i, k, l - 1) + d(k, j, l - 1))$
```

Comments:

- It was important that ℓ was in the outermost loop, since that was the only argument that was guaranteed to decrease; i and j could be swapped.
- In a lot of incorrect solutions there were variables $(\ell, k, \dots)$ which were not bounded to a loop. Note that ℓ is not part of the input and so it should be a loop variable.

(d) The running time of the algorithm on a graph with n vertices is $O(n^3 \log n)$

Comments:

- Every running time below $n^3 \log n$ is incorrect. Higher running times which do not account for a tight upper bound of l, were granted only partial credit.

Room Rentals (15 points + 15 extra-credit points)

9. You have two rooms to rent out. There are n customers interested in renting the rooms. The i^{th} customer wishes to rent one room (is happy with either room you have) for $t[i]$ days and is willing to pay $bid[i]$\$ for his/her entire stay.

   Customer requests are non-negotiable in that they would not be willing to rent for a shorter or longer duration.

   Devise a dynamic programming algorithm to determine the maximum profit that you can make from the customers over a period of D days.

   (Hint: two knapsacks?)

   (a) Briefly and precisely define the subproblems.

   Solution:
   $P(d_1, d_2, i) :=$ the maximum profit obtainable with d_1 remaining days for room 1 and d_2 remaining days for room 2 using the first i customers.

   Common Mistakes:

   - One common incorrect subproblem was having two recurrences, one for each room. The problem with this approach is that the two subproblems have exactly the same recurrences, and thus, the same solutions; this allows the same customer to be placed in both rooms.
   - Another common incorrect subproblem was having a recurrence that only keeps track of one room up to $2D$ days. However, if a customer wants to stay for 4 days and there are 2 days left in both rooms, then this approach of combining the rooms into one incorrectly suggests that the customer can stay.
   - Some students seemed unclear on what a subproblem was. We were not looking for a recurrence, or an explanation of how to *solve* the subproblem. (Writing a correct recurrence without defining the subproblem received no credit in this part). The definition of a subproblem is just the smaller "problem" that we wish to solve (e.g. here we are trying to maximize profit in a specialized case).

   (b) Write the recurrence relation between the subproblems.
   Solution:

$$P(d_1, d_2, i) = \max(bid[i] + P(d_1 - t[i], d_2, i-1), bid[i] + P(d_1, d_2 - t[i], i-1), P(d_1, d_2, i-1))$$

   Intuitively, we can either give customer i the first room, give customer i the second room, or not give customer i either room, which corresponds to the quantities being maximized over.

   Common Mistakes: See the common mistakes section in part (a) for common incorrect subproblems (and thus incorrect recurrence relations).

(c) *(Extra Credit, Attempt at the end (15 points))* Suppose every customer has a specific start date $start[i]$ and end date $end[i]$ between which he/she is interested in renting.

How would you modify the algorithm, and the definition of your subproblems? (Give a succinct but precise description of the modifications and the subproblems. Proof of correctness & runtime analysis not needed)

Solution:
Sort the customers by their $end[i]$ day, so that we have:

$$end[1] \leq \cdots \leq end[n]$$

For convenience, assume that $end[i] \leq D$ for all costumers i.

Define $P(d_1, d_2, i)$ to be the maximum profit obtainable using d_1 days for room 1, d_2 days for room 2, and the first i customers.

Then we have the recurrence:

$$P(d_1, d_2, i) = \begin{cases} \max \begin{cases} P(start[i], d_2, i-1) + bid[i], \\ P(d_1, start[i], i-1) + bid[i], \\ P(d_1, d_2, i-1) \end{cases} & \text{, if } end[i] \leq d_1, d_2 \\ \max \begin{cases} P(start[i], d_2, i-1) + bid[i], \\ P(d_1, d_2, i-1) \end{cases} & \text{, if } d_2 < end[i] \leq d_1 \\ \max \begin{cases} P(d_1, start[i], i-1) + bid[i], \\ P(d_1, d_2, i-1) \end{cases} & \text{, if } d_1 < end[i] \leq d_2 \\ P(d_1, d_2, i-1) & \text{, if } d_1, d_2 < end[i] \end{cases}$$

This can be simplified into the following:

$$P(d_1, d_2, i) = \max \begin{cases} \begin{cases} P(start[i], d_2, i-1) + bid[i], & \text{if } end[i] \leq d_1 \\ -\infty & \text{otherwise} \end{cases} \\ \begin{cases} P(d_1, start[i], i-1) + bid[i], & \text{if } end[i] \leq d_2 \\ -\infty & \text{otherwise} \end{cases} \\ P(d_1, d_2, i-1, true) \end{cases}$$