CS/ECE 374 A (Spring 2020)
Final Exam Solutions

Problem 1: [20 PTS] *Short questions.*

(a) [5 PTS] Draw an NFA for the following regular expression:

$$(11 + 0)^*((01)^*(10)^* + (11)^*(00)^*).$$

(Justification is not required; ε-transitions are allowed.)

(b) [5 PTS] Draw a DFA for the following language

$$L = \{x \in \{0,1\}^* \mid x \text{ contains the substring 010 an even number of times}\}.$$

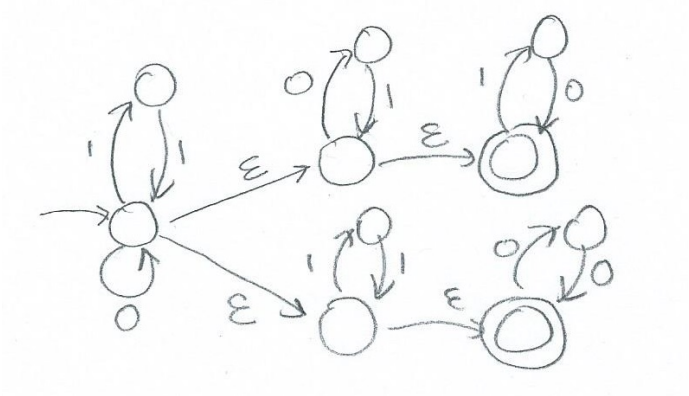
(Justification is not required; occurrences of the substring may overlap, and remember that zero is an even number.)

(c) [5 PTS] Given a NFA M, consider the problem of deciding whether $L(M) = \emptyset$. Is this problem in the class P? Justify your answer.

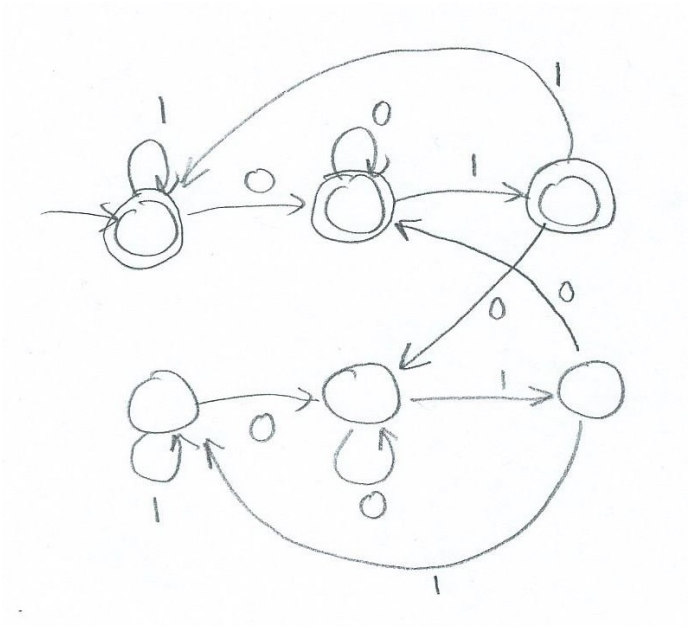
(d) [5 PTS] Given a directed graph $G = (V, E)$ where all the edges have positive weights, two vertices $s, t \in V$, and given a number W, consider the problem of deciding whether there exists a path from s to t with total weight *at most* W. Is this problem NP-complete? Explain your answer (and state your assumptions).

Solution:

(a)



(b)



(c) Yes. The problem is equivalent to deciding whether any accepting state is reachable from the start state. We can find all states reachable from the start state by BFS or DFS on the graph formed by the transition diagram. The problem can be solved in time linear in the size of the graph ($O(|Q||\Sigma|)$), which is polynomial.

(d) No, assuming P$\neq$ NP. (Or "we don't know", since it is equivalent to the open question P$\neq$ NP.)

The problem is in P, since we can run Dijkstra's algorithm to compute the shortest path from s to t and check whether the path has total weight at most W. Dijkstra's algorithm takes polynomial time. If the problem is both in P and is NP-complete, then that would imply P = NP. (Conversely, if P = NP, then any problem in P that is not $\emptyset$ or Σ^* would be NP-complete by trivial polynomial-time reductions.)

Problem 2: [16 PTS] Consider the following two problems (where M denotes a Turing machine, $L(M)$ denotes the language accepted by M, and $\langle M \rangle$ denotes the encoding of M, and $\langle M, n \rangle$ denotes the encoding of M and an integer n):

$L_1 = \{\langle M \rangle \mid L(M)$ is in P (i.e., is solvable in polynomial time)$\}$

$L_2 = \{\langle M, n \rangle \mid M$ does not terminate within n^{2020} steps for some input binary string of length $n\}$.

(a) [10 PTS] Prove that L_1 is undecidable. (Note: you may not use Rice's theorem.)

(b) [6 PTS] Is L_2 decidable? Prove your answer.

Solution:

(a) For the sake of contradiction, suppose there is an algorithm DECIDE-PART-A that correctly decides L_1.

Pick some language L_{bad} that is decidable but known not to be in P (for example, the language corresponding to the problem of testing equivalence of two generalized regular expressions, as mentioned in Jeff's book). Let M_{bad} be the TM that decides L_{bad}.

We now solve the halting problem as follows (returning "true" means "accept", and "false" means "reject"):

DECIDE-HALT($\langle M, w \rangle$):

Encode the following Turing machine M':

> $M'(x)$:
> run M on w (note: this step may or may not terminate)
> run M_{bad} on x (note: this step always terminates)
> if M_{bad} accepts x then return true else return false

If DECIDE-PART-A($\langle M' \rangle$) then return false else return true

DECIDE-HALT($\langle M, w \rangle$) always terminates.

We prove: M halts on input $w \iff$ DECIDE-HALT($\langle M, w \rangle$) returns true.

$\implies$ Suppose M halts on w. Then $L(M') = L_{\text{bad}}$, which is not in P. So, DECIDE-PART-A($\langle M' \rangle$) returns false. So, DECIDE-HALT($\langle M, w \rangle$) returns true.

$\impliedby$ Suppose M does not halt on w. Then $L(M') = \emptyset$, which is obviously in P. So, DECIDE-PART-A($\langle M' \rangle$) return true. So, DECIDE-HALT($\langle M, w \rangle$) returns false.

Thus, DECIDE-HALT correctly solves the halting problem. But the halting problem is undecidable: contradiction.

[*Note*: if you pick an NP-complete problem for L_{bad}, then we won't know for sure that L_{bad} is not in P (since P $\neq$ NP is not yet proven)...]

[*Alternative solution*: We can alternatively reduce from ACC $= \{\langle M, w \rangle \mid M \text{ accepts } w\}$ instead of the halting problem. The proof is almost identical, except that in M', after running M on w, if M terminates and rejects w, then M' immediately returns false; otherwise, we proceed as before...]

(b) Answer: **decidable**.

On input $\langle M, n \rangle$, we just go through all 2^n binary strings x of length n, and run M on x up to a limit of n^{2020} steps, and we output yes iff one of these runs does not finish in n^{2020} steps. This algorithm always terminate with the correct answer (running in $O(n^{2020}2^n)$ time).

Problem 3: [16 PTS] Let $\Sigma = \{0, 1\}$. We are given a context-free grammar $G = (V, \Sigma, P, S)$ in Chomsky normal form, i.e., all production rules in P are of the form $A \to BC$ or $A \to a$ for some variables $A, B, C \in V$ and terminal $a \in \Sigma$. (There are no productions of the form $A \to \varepsilon$.) We are also given an integer n. Consider the problem of deciding whether there exists a string of length exactly n that is in $L(G)$ (the language generated by G).

Here is one approach based on dynamic programming: For each variable $A \in V$ and each $k = 1, \ldots, n$, we define $f(A, k)$ to be true iff A generates a string of length exactly k. We then have the recursive formula for $k \geq 2$:

$$f(A,k) \;=\; \bigvee_{j=1}^{k-1} \bigvee_{A \to BC \text{ is a rule}} (f(B,j) \wedge f(C,k-j)).$$

For the base case $k = 1$, we have $f(A, 1) =$ true iff $A \to a$ is a rule for some $a \in \Sigma$.

(a) [10 PTS] Give pseudocode to solve the problem using the above formula (which you may use without proof), and analyze its running time as a function of n and $|V|$ and $|P|$. The output should be "true" or "false".

(b) [6 PTS] Now, suppose we change the problem to the following: decide whether there exists a string with exactly n 0's and exactly n 1's that is generated by G. Give a modified definition of f and the recursive formula (including base cases), and bound the running time. Do not give pseudocode for this part. (Justification of correctness is not required.)

Solution:

(a)

```
1.  for each A ∈ V do f[A,1] = true iff there is a rule of the form A → a
2.  for k = 2 to n do
3.      for each A ∈ V do {
4.          f[A,k] = false
5.          for each rule of the form A → BC do
6.              for each j = 1 to k − 1 do
7.                  if f[B,j] ∧ f[C,k−j] then f[A,k] = true
        }
8.  return f[S,n]
```

Run time. Lines 5–7 take time $O(kp_A)$ where p_A is the number of rules of the form $A \to BC$. Summing over all A gives $O(k|P|)$. Thus, the total time for lines 2–7 is $O(k|P| + |V|)$. Summing over all k gives total time $O(n^2|P| + n|V|)$.

(Note: W.l.o.g., we may assume $|V| \leq |P|$, and the expression would then simplified to $O(n^2|P|)$.)

(b) We redefine $f(A, k, k')$ to be true iff A generates a string with exactly k 0's and k' 1's. We want $f(S, n, n)$. The recursive formula is changed to the following: for $k + k' \geq 1$,

$$f(A,k,k') \;=\; \bigvee_{j,j':\ 1 \leq j+j' < k+k',\ j \leq k,\ j' \leq k'} \ \bigvee_{A \to BC \text{ is a rule}} (f(B,j,j') \wedge f(C,k-j,k'-j')).$$

For the base cases, $f(A, 1, 0) =$ true iff $A \to 0$ is a rule; $f(A, 0, 1) =$ true iff $A \to 1$ is a rule.

We evaluate in increasing order of $k + k'$. With additional loops over k' and j', the run time increases to $O(n^4|P| + n^2|V|)$. (Again, the second term may be dropped.)

Problem 4: [16 PTS] Given a directed graph $G = (V, E)$ where each edge is colored red or blue, and given an integer k, we want to decide whether there exists a cycle of length at least k such that the colors of the edges alternate between red and blue (i.e., the 1st, 3rd, 5th, etc. edges are red, and the 2nd, 4th, 6th, etc. edges are blue, and the length is even). Recall that repeated vertices are not allowed in a cycle. Call this problem ALT-CYCLE. You will prove that ALT-CYCLE is NP-complete.

(a) [5 PTS] Prove that ALT-CYCLE is in NP.

(b) [11 PTS] Prove that ALT-CYCLE is NP-hard.
(Note: You may use the fact that HAMILTONIAN-CYCLE for directed graphs is NP-complete. Partial credit may be given for just following the format and main steps of an NP-completeness proof.)

Solution:

(a) **Certificate:** a cycle C.
Clearly, C has polynomial ($O(n)$) size.

Conditions to verify: C has length at least k and the colors in C alternate between red and blue (and $C \subseteq E$, and C does not repeat vertices).
We can check this by scanning the edges in C in linear ($O(n)$) time, which is polynomial.

(b) We give a polynomial-time reduction from HAMILTONIAN-CYCLE (directed version) to ALT-CYCLE.

The reduction: Given a directed graph $G = (V, E)$ (input to HAMILTONIAN-CYCLE), we construct a new directed graph $G' = (V', E')$ where each edge is colored red/blue, and an integer k (input to ALT-CYCLE), as follows:

- For each $v \in V$, we add two vertices $v^\sharp$ and $v^\flat$ to V', and we add a red edge $(v^\sharp, v^\flat)$ to E'.
- For each $(u, v) \in E$, we add a blue edge $(u^\flat, v^\sharp)$ to E'.
- We let $k = 2n$.

This construction from G to G' clearly takes polynomial time.

Correctness: We show that there exists a Hamiltonian cycle in $G \iff$ there exists a cycle in G' of length at least k with alternating colors.

Proof of ($\Longrightarrow$): Suppose G has a Hamiltonian cycle $\langle v_1, v_2, \ldots, v_n, v_1\rangle$. Then $\langle v_1^\sharp, v_1^\flat, v_2^\sharp, v_2^\flat, \ldots, v_n^\sharp, v_n^\flat, v_1^\sharp$ is a cycle in G', and colors alternate between red and blue, and the length of the cycle is $2n = k$.

Proof of ($\Longleftarrow$): Suppose π is a cycle in G', with alternating colors, and length equal to k. Since π alternates between red and blue, it must be of the form $\langle v_1^\sharp, v_1^\flat, v_2^\sharp, v_2^\flat, \ldots, v_{k/2}^\sharp, v_{k/2}^\flat, v_1^\sharp\rangle$, where $v_1v_2, \ldots, v_{k/2}v_1 \in E$. Thus, $\langle v_1, v_2, \ldots, v_{k/2}, v_1\rangle$ forms a closed walk in G. Since π does not have repeated vertices, $v_1, \ldots, v_{k/2}$ are distinct. Thus, $\langle v_1, v_2, \ldots, v_{k/2}, v_1\rangle$ is a cycle in G. Since $k = 2n$, it is a Hamiltonian cycle in G.

Problem 5: [16 PTS] We are given a connected undirected graph $G = (V, E)$ with n vertices and m edges, and two distinct vertices $s, t \in V$. We are also given an ordering of the edges $e_1, \ldots, e_m$. Imagine deleting the edges in the given order; at some moment in time, t will become disconnected from s. We want to find the moment when that happens. More precisely, we want to find the smallest index i such that there is no path from s to t in the subgraph $G_i = (V, E - \{e_1, \ldots, e_i\})$.

Describe the fastest algorithm you can think of to solve this problem. (Note: There are several possible solutions; one way is to modify a known algorithm for minimum spanning trees, and another way is to directly use minimum spanning trees as a black box. If correctness of your algorithm is not obvious, justification may be required. Partial credit will be given for slow but correct algorithms.)

First Solution: For each $i = 1, \ldots, m$, we can test whether t is reachable from s in G_i by BFS/DFS in $O(m + n) = O(m)$ time (we have $m \geq n - 1$ since G is connected). The smallest i for which the answer is no gives the answer. The total time is $O(m^2)$.

Second Solution: Use binary search. Let i^* denote the unknown index. For a given i, testing whether $i^* > i$ is equivalent to testing whether t is reachable from s in G_i, which can be done by BFS/DFS in $O(m)$ time. We can find i^* by binary search using $O(\log m)$ tests. The total time is $O(m \log m) = O(m \log n)$.

Third Solution:

1. assign weight $-i$ to each edge e_i
2. compute MST T
3. let π be the path from s to t in T
4. let e_i be the edge in π with the smallest i, and return i

Line 2 takes $O(n \log n + m)$ time by Prim's algorithm with Fibonacci heaps (or better with some of the more advanced MST algorithms cited in the slides). Line 3 takes $O(n)$ time (by following parent pointers, assuming s is made the root). Line 4 takes $O(n)$ time by a linear scan over π. The total time is therefore $O(n \log n + m)$ (which is strictly better than the previous solution for graphs not too sparse).

To prove correctness, let i be the index returned by the algorithm.

- **Claim**: *There is a path from s to t in G_{i-1}.*
  Proof: Every edge e_j in π has $j \geq i$. So, π lies in $E - \{e_1, \ldots, e_{i-1}\}$, i.e., in G_{i-1}.

- **Claim**: *There is no path from s to t in G_i.*
  Proof: $T - \{e_i\}$ has two connected components; call them S and $V - S$. We know that $s \in S$ and $t \in V - S$, or vice versa. By a known fact from class (also used in HW10.2), e_i must be the smallest-weight edge between S and $V - S$. Thus, every edge e_j between S and $V - S$ has $j \leq i$ and so lies in $\{e_1, \ldots, e_i\}$. So, s and t are in different components in $G_i = (V, E - \{e_1, \ldots, e_i\})$.

Fourth Solution: We imitate Kruskal's algorithm.

Start with the empty graph $(V, \emptyset)$. For each $i = m, \ldots, 1$, insert e_i to the graph. Maintain the connected components. Stop when s and t lies in the the same components. Return $i+1$.

Inserting an edge $e_i = uv$ requires merging two components (a "union" operation), if u and v are in different components. Checking whether two vertices are in the same component is a "find" operation. By known "union-find" data structure, we can implement the algorithm in $O(m\alpha(n))$ time (or more accurately, $O(m\alpha(m, n))$), where $\alpha(\cdot)$ is the inverse Ackermann function. This is actually faster than $O(m \log n)$ (or even $O(n \log n + m)$) (although in the slides, we have only mentioned logarithmic-time union-find data structures, which gives $O(m \log n)$ time). Note that unlike the original Kruskal's algorithm, we don't need to sort first, so there is no $O(m \log n)$ extra cost.

Fifth Solution (Rough Sketch): There is actually a linear ($O(m)$) time algorithm (this is deterministic, and does not use advanced MST algorithms). We speed up the earlier binary-search solution. We first test whether $i^* > m/2$, in $O(m)$ time. If $i^* > m/2$, we can remove the edges $e_1, \ldots, e_{m/2}$ from G. On the other hand, if $i^* \le m/2$, we can contract the edges $e_{m/2}, \ldots, e_m$ in G ("contracting" an edge uv means we glue the two vertices u and v together into one). We omit the details... In either case, the number of edges is decreased by half. The total time is $O(m + m/2 + m/4 + \cdots) = O(m)$.

[*Grading Note*: max 5 pts for $O(m^2)$, max 13 pts for $O(m \log n)$, max 16 pts for $O(n \log n + m)$ or $O(m\alpha(n))$ or better, and max 16 pts plus 3 bonus pts for a deterministic $O(m)$ algorithm as in the fifth solution.]

Problem 6: [16 PTS] Consider the following problem: given a sequence of n positive numbers $A = \langle a_1, \ldots, a_n \rangle$ and a given value W, we want to partition A into the smallest number k^* of blocks (i.e., contiguous subsequences) such that each block has sum at most W.

For example: if $A = \langle 2, 7, 1, 4, 2, 2, 3, 5 \rangle$ and $W = 10$, then one optimal solution is to divide A into $k^* = 3$ blocks $\langle 2, 7 \rangle$, $\langle 1, 4, 2, 2 \rangle$, and $\langle 3, 5 \rangle$ (the block sums 9, 9, and 8 are all at most 10).

Consider the following two greedy strategies:

Strategy I: at each iteration, select the longest block $\langle a_i, \ldots, a_j \rangle$ (i.e., the block with the largest $j - i + 1$) that has sum at most W and does not overlap with previously selected blocks.

Strategy II: at each iteration, find the longest prefix $\langle a_1, \ldots, a_i \rangle$ that has sum at most W, select the block $\langle a_1, \ldots, a_i \rangle$, remove these elements, and repeat.

(a) [4 PTS] Show that Strategy I does not always correctly find an optimal solution, by providing a counterexample.

(b) [3 PTS] What is the best running time to implement Strategy II? Briefly explain.

(c) [9 PTS] Formally prove that Strategy II always outputs an optimal solution.
(Hint: if an optimal solution does not use the block chosen by the algorithm in the first iteration, could you modify the optimal solution to use that block?)

Solution:

(a) One counterexample: $\langle 2, 1, 1, 2\rangle$ with $W = 3$. Strategy I would pick $\langle 1, 1\rangle$, and then $\langle 2\rangle$ and $\langle 2\rangle$, which uses 3 blocks. But a better solution is to pick $\langle 2, 1\rangle$ and $\langle 1, 2\rangle$, which uses 2 blocks.

(b) $O(n)$ time. We just scan the input sequence from left to right, and keep track of the current prefix sum (by adding the current element to the current sum). When the current sum exceeds W, we start a new block and reset the current sum.

(c) Let $B_1, \ldots, B_k$ be the blocks, from left to right, in the solution computed by the algorithm.

We claim that for each $\ell \in \{0, \ldots, k^*\}$, there is an optimal solution O^* with blocks $B_1^*, \ldots, B_{k^*}^*$, from left to right, such that $B_1 = B_1^*, \ldots, B_\ell = B_\ell^*$.

The statement is trivially true for the base case $i = 0$.

Suppose the statement is true for $\ell - 1$. Thus, $B_1 = B_1^*, \ldots, B_{\ell-1} = B_{\ell-1}^*$. Suppose $B_\ell \neq B_\ell^*$. Say $B_1, \ldots, B_{\ell-1}$ cover $a_1, \ldots, a_{s-1}$. Say B_ℓ^* is $\langle a_s, \ldots, a_{i^*}\rangle$ and B_ℓ is $\langle a_s, \ldots, a_i\rangle$. By the definition of greedy strategy II, we know that $i \geq i^*$, i.e., B_ℓ contains B_ℓ^*. Suppose B_ℓ contains $B_\ell^*, \ldots, B_m^*$ and partially overlaps with B_{m+1}^*. In the optimal solution O^*, we can remove the blocks $B_\ell^*, \ldots, B_m^*$, add the new block B_ℓ, and remove elements from B_{m+1}^* that are covered by B_ℓ. Then we still have a partition in which all blocks have sum at most W. Thus, the new solution is still feasible. And the number of blocks does not increase (since the decrease is $m - \ell^* \geq 0$). And the new solution contains B_ℓ. Hence, there exists an optimal solution using $B_1, \ldots, B_\ell$. This completes the induction proof.

The conclusion follows by setting $\ell = k^*$.

[*Grading Note*: no pts deducted if induction is not explicitly used...]