**CS 344: Design and Analysis of Computer Algorithms** **Rutgers: Spring 2021**

# Homework #5

Deadline: Monday, May 03, 11:59 PM

*Name: Letao Zhang* *Extension: Yes*

## Homework Policy

- If you leave a question completely blank, you will receive 25% of the grade for that question. This however does not apply to the extra credit questions.
- You are allowed to discuss the homework problems with other students in the class. **But you must write your solutions independently.** You may also consult all the materials used in this course (video recordings, notes, textbook, etc.) while writing your solution, but no other resources are allowed.
- Do not forget to write down your name and whether or not you are using one of your two extensions. Submit your homework on Canvas.
- Unless specified otherwise, you may use any algorithm covered in class as a "black box" – for example you can simply write "use Ford-Fulkerson's algorithm to find a maximum flow of the input network in $O(m \cdot F)$ time". You are **strongly encouraged to use graph reductions** instead of designing an algorithm from scratch whenever possible (even when the question does not ask you to do so explicitly).
- Remember to always **prove the correctness** of your algorithms and **analyze their running time** (or any other efficiency measure asked in the question).
- The "Challenge yourself" and "Fun with algorithms" are both extra credit. These problems are significantly more challenging than the standard problems you see in this course (including lectures, homeworks, and exams). As a general rule, only attempt to solve these problems if you enjoy them.

---

**Problem 1.** You are given an $n \times n$ matrix and a set of $k$ cells $(i_1, j_1), \ldots, (i_k, j_k)$ on this matrix. We say that this set of cells can **escape** the matrix if: (1) we can find a path from each cell to any arbitrary *boundary cell* of the matrix (a path is a sequence of *neighboring* cells, namely, top, bottom, left, and right), (2) these paths are all *disjoint*, namely, no cell is used in more than one of these paths.

Design an $O(n^3)$ time algorithm that given the matrix and the input cells, determines whether these cells can escape the matrix (together) or not. **(25 points)**

**Solution.** *Algorithm.* Find the maximum flow is to find the maximum number of edge-disjoint paths from a vertex $s$ to a vertex $t$ in a given graph $G = (V, E)$. Simply turn $G$ into a network by assigning capacity 1 to every edge (and if $G$ is originally undirected, add both direction of edges to make it directed). Then find the maximum flow from $s$ to $t$ in this network and return the edges with non-zero flow as the edges of the paths, then output the cells.

*Proof of Correctness.* Firstly, any $s - t$ flow of value $k$ in this network corresponds to a collection of edge-disjoint paths $P_1$, ..., $P_k$; this is because each edge can only carry one unit of flow in the network and so any of the $k$ units of flow starting from $s$ and ending in $t$ should follow a path, edge-disjoint from the rest, from$s$ to $t$. This proves that the solution is feasible. Secondly, any collection of edge-disjoint paths$P_1$, ..., $P_\ell$in the graph implies a flow of value $\ell$ in the network; simply route one unit of flow across each path and since they are edge-disjoint, no edge of the network carry more than 1 unit of flow which is its capacity. This then means there is a one-to-one mapping between flows and edge-disjoint paths and thus by returning the maximum flow, we are also returning maximum size collection of edge-disjoint paths. This proves that the solution is optimal.

*Runtime Analysis.* Vertex disjoint path algorithm on the surface takes $O(n^2 \cdot k)$ time where k is the number of input cells which can be as large as $n^2$. If $k > 4n$, the answer is always NO because there is no way to send more 4n cells to outside. So, when $k > 4n$, you can simply output NO and otherwise we have $k = O(n)$ and run the algorithm which will take $O(n^3)$ time.

---

**Problem 2.** You are given an undirected *bipartite* graph $G$ where $V$ can be partitioned into $L \cup R$ and every edge in $G$ is between a vertex in $L$ and a vertex in $R$. For any integers $p, q \geq 1$, a $(p, q)$*-factor* in $G$ is any subset of edges $M \subseteq E$ such that no vertex in $L$ is shared in more than $p$ edges of $M$ and no vertex in $R$ is shared in more than $q$ edges of $M$.

Design an $O((m + n) \cdot n \cdot (p + q))$ time algorithm for outputting the size of the *largest* $(p, q)$-factor of any given bipartite graph.

**(25 points)**

**Solution.** *Algorithm.* We simply can solve bipartite matching using network flow. We now use a graph reduction to solve the bipartite matching problem using any algorithm for the network flow problem. In order to do this, we should show that given any bipartite graph $G = (V, E)$, we can turn it into a flow network $G' = (V', E')$ with a source $s$ and sink $t$, such that finding the maximum flow in $G'$ allows us to find a maximum matching in $G$.

*Proof of Correctness.* We first have to prove that the set of edges $M$ found by the algorithm is indeed a matching and then prove its size is maximum.

1) $M$ is a matching: For $M$ not to be a matching, a vertex $v$ should be shared by more than one edge of $M$. However, this cannot happen because if $v \in L$, then only one unit of flow can ever each $v$ and so at most one outgoing edge of $v$ can have flow equal to one and hence be added to $M$; similarly, if $v \in R$, then only one unit of flow can ever go out of $v$ and so at most one incoming edge of $v$ can have flow equal to one and be added to $M$.

2) $M$ is a maximum matching: We prove that any flow $f$ in $G'$ corresponds to a matching $M$ in $G$ with size of $M$ equal to the value of flow $f$ and vice versa. This then implies that maximum flow corresponds to a maximum matching and thus $M$ is a maximum matching. By the above part, we know that for any flow $f$ in $G'$ there is a matching $M$ of the same size in $G$. We now prove the other direction.Fix any matching $M$ in $G$ and consider the flow $f$ where for any edge$u, v$ in $M$, we add a flow path $f(s, u) = 1$, $f(u, v) = 1$, and $f(v, t) = 1$ to our flow $f$. This definition of $f$ clearly satisfies the preservation flow. Moreover, since by definition of a matching $M$, the vertices $u$, $v$ only appear once in the matching, we have that $f$ also satisfies the capacity constraints and hence is indeed a flow. The value of this flow also is equal to $\sum_{v \in L} f(s, v)$ which is equal to the size of $M$ by definition.

*Runtime Analysis.* We can create the network above in $O(m + n)$ time by traversing the edges of graph $G$ directly. Then run Ford-Fulkerson algorithm and find this maximum flow in $O(m \cdot F)$ time where $F$ is the value of maximum flow. As the value of maximum flow in our network is at most $p + q$, this gives an algorithm with runtime $O(n \cdot (p + q))$ for the bipartite matching problem. Again, an important reminder that we should always state the runtime of our algorithms in terms of parameters of the original input, so the runtime here is $O((m + n) \cdot n \cdot (p + q))$.

---

**Problem 3.** Given an undirected graph $G = (V, E)$ and an integer $k \geq 2$, a $k$-coloring of $G$ is an assignment of $k$ colors to the vertices of $V$ such that no edge in $E$ has the same color on both its endpoints.

(a) Design a poly-time *algorithm for solving* the decision version of the 2-coloring problem: Given a graph $G = (V, E)$ output *Yes* if $G$ has a 2-coloring and *No* if it does not. **(15 points)**

**Solution.** *Algorithm.* Deciding if a given graph is 2-colorable is equivalent to determining whether it is a bipartite graph, which only takes polynomial time.

Pick an arbitrary vertex $s \in V$. Run BFS (for shortest path) from $s$ and define the layers $L_1$, $L_2$, ..., of vertices where $L_i := v \in V|d[v] = dist(s, v) = i$.

For each vertex $v$, stored which of these sets it belongs to.Iterate over the edges $E$ of $G$ and for each edge $e = (u, v) \in E$, if both $u$ and $v$ belong to the same level $L_i$,output G is not bipartite and terminate. Otherwise, if the for-loop never terminated, output $G$ is bipartite.

*Proof of Correctness.* Part one: If $G$ is bipartite, then the algorithm also outputs bipartite. Assume toward the contradiction that the algorithm has output $G$ is not bipartite. For this to happen, there should be an edge $(u, v)$ such that $dist(s, u) = dist(s, v) = i$. Pick the shortest paths $P^u$ and $P^v$ from $s$ to $u$ and $v$, respectively. Let $j$ be the last index where $P^u_j = P^v_j$ and suppose $w$ is the $j$-th vertex of these paths.

This means that there is a cycle $C$ in $G$ starting from $w$ going to $u$, taking the edge $(u, v)$, and then going back from $v$ to $w$. Moreover, the length of this cycle is an odd number, in particular $2 \cdot (i - j) + 1$.Recall that we assumed $G$ is bipartite and without loss of generality we can assume $w \in L$. Then, the next vertex of the cycle $C$ should be in $R$, the next one in $L$, and so on and so forth. But because $C$ has an odd length, this forces $w$ to also be in $R$, a contradiction. This means that $G$ cannot be a bipartite graph if the algorithm outputs not bipartite.

If the algorithm outputs bipartite, then $G$ is also bipartite. Consider adding $s$ to $L$, $L_1$ to $R$, $L_2$ to $L$, $L_3$ to $R$, and so on and so forth. We prove that all edges of $G$ are between the sets $L$ and $R$. Since $L_1$, $L_2$, ...., are computed by BFS, we know that any other edge (u, v) in the graph should be either inside one layer, or from one layer to one after or before it (otherwise, there is a shorter path starting from $s$ to one endpoint of the edge, a contradiction). Moreover, since we checked in the algorithm that there is no edge inside one layer, all edges of $G$ are between one layer and the next.Thus, they are all between $L$ and $R$ by the construction above.
*Runtime Analysis.* Running BFS takes $O(n + m)$ time. Marking the layer of each set also takes $O(n)$ time and checking the edges require another $O(m)$ time. So, in total the runtime is $O(n + m)$.

---

(b) Design a poly-time *verifier* for the decision version of the $k$-coloring problem for any $k \geq 2$: Given a graph $G = (V, E)$ and $k$ as input, output *Yes* if $G$ has a $k$-coloring and *No* if it does not. Remember to specify exactly what type of a proof you need for your verifier. **(10 points)**

**Solution.** *Rerifer.* If the graph is k-colorable ($Yes$), we can simply use any k-coloring of the graph as a proof. The following is a verifier $V$ for k-coloring problem. 1) Check that $v$ includes $\leq 2$ colors. 2)Color each node of $G$ as specified by $v$. 3)For each node, check that it has a unique color from each of its neighbors. 4. If all checks pass, *Yes*; otherwise, *No*.

*Proof of Correctness.* Let $v$ be the maximum degree of all vertices in graph $G$. Then $G$ can be colored with $v + 1$ colors in polynomial time. Simple greedy-like arguments will work in this case. We color the vertices in any order. When coloring a vertex, since it has at most $v$ neighbors, there's at least one color that will not yield any edge if colored on this vertex. So we can continue this process until all vertices are colored.

*Runtime Analysis.* The graph has m edges, which makes the runtime be $O(m)$. Then we choose 2 vertices to receive the same 2 color, and the rest of the vertices each receive their own color taking $O(n)$. We can try all such candidate colorings in polynomial time and see whether any of them is valid. The runtime of this algorithm is $O(m + n)$, which is polynomial time..

---

**Problem 4.** Prove that each of the following problems is NP-hard and for each problem determine whether it is also NP-complete or not.

(a) **One-Fourth-Path Problem:** Given an undirected graph $G = (V, E)$, does $G$ contain a path that passes through *at least one forth* of the vertices in $G$? **(8 points)**

**Solution.** *Reduction.* We prove this problem is NP-hard by a reduction from the undirected $s - t$ Hamiltonian path problem. Given an arbitrary graph $G$, let $H$ be a graph consisting of two disjoint copies of $G$, with no edges between them; call these copies $G_1$ and $G_2$. Claim that $G$ has a Hamiltonian cycle if and only if $G$ contains a path..

*Proof of Correctness.* Suppose $G$ has a Hamiltonian cycle $C$ and contains a path. Let $C_1$ be the corresponding cycle in $G_1$. $C_1$ contains one forth of the vertices of $H$. On the other hand, suppose $H$ has a Hamiltonian cycle $C$ and contains a path. Because there are no edges between the subgraphs $G_1$ and $G_2$, this cycle must lie entirely within one of these two subgraphs. $G_1$ and $G_2$ each contain one forth of the vertices of $H$, so $C$ must also contain exactly one forth of the vertices of $H$, and thus is a Hamiltonian cycle in either $G_1$ or $G_2$. Knowing that $G_1$and $G_2$ are just copies of $G$. We conclude that $G$ contains a path.
*Runtime Analysis.* This can be done in polynomial time, which is $O(m + n)$. By definition we know that undirected $s - t$ Hamiltonian path is NP-hard, then we say that this problem is NP-hard that has a polynomial-time algorithm for implying that P = NP. Since it is in NP and NP-hard, we can say that NP-complete.

---

(b) **Two-Third 3-SAT Problem:** Given a 3-CNF formula $\Phi$ (in which size of each clause is *at most* 3), is there an assignment to the variables that satisfies at least 2/3 of the clauses? **(8 points)**

**Solution.** *Reduction.* We prove this problem is NP-hard by a reduction from 3-SAT problem. If $\Phi$ is satisfiable then there should exists some assignment $y$ to the variables where $\Phi(y) = 1$.We can use any such $y$ as a proof.

*Proof of Correctness.* Let $C$ be any given circuit and $\Phi$ be the resulting 3-CNF formula in the reduction.We prove that $C$ is satisfiable if and only if $\Phi$ is satisfiable. This will immediately imply the correctness. In order to do this, we first prove that $\Phi$ and $C$ are logically equivalent. To do this, we need to show that the set of clauses introduced for each gate work exactly the same as the gate itself.

If $C$ is satisfiable, then $\Phi$ is satisfiable also: Pick a satisfying assignment $x$ to $C$ and consider every wire of this circuit. Let $y$ be the assignment of these wires to variables in $\Phi$. By the above part, $C$ and $\Phi$ are logically equivalent and thus $y$ satisfies every clause in $\Phi$. Finally, since for the output wire we have a singleton clause z(which is equivalent to $C(x)$ which in turn is 1), the $\Phi(y) = 1$. This means $\Phi$ is satisfiable.

If $\Phi$ is satisfiable, then $C$ is satisfiable also. Pick a satisfying assignment $y$ to $\Phi$ and consider the variables assigned to the input wires. Let $x$ be the assignment of these input wires in $C$. By the above part, $C$ and $\Phi$ are logically equivalent and so every wire of circuit $C$ on the input $x$ will get the same value as the corresponding variable $y$ in $\Phi$. This in particular means that the output wire gets the value 1 on the input $x$ and thus $C(x) = 1$. This means $C$ is satisfiable.

*Runtime Analysis.* We show that if we have a poly-time algorithm for 3-SAT the above reduction runs in poly-time. This is true because the size of $\Phi$ is at most a constant factor larger than size of the circuit and we create $\Phi$ in linear-time from $C$. Thus a poly-time algorithm for 3-SAT on $\Phi$ implies a poly-time algorithm for Circuit-SAT on $\Phi$.

This concludes the proof of NP-hardness of the 3-SAT problem. Given that we already proved 3-SAT is also in NP, this means 3-SAT is NP-complete.

---

(c) **Negative-Weight Shortest Path Problem:** Given an undirected graph $G = (V, E)$, two vertices $s, t$ and *negative* weights on the edges, what is the weight of the shortest path from $s$ to $t$? **(9 points)**

**Solution.** *Reduction.* A Hamiltonian cycle in an undirected graph $G = (V, E)$ is a cycle that passes through every vertex.The Hamiltonian cycle problem asks whether a given undirected graph has any Hamiltonian cycle or not Hamiltonian cycle can also be shown to be NP-hard. We prove this problem is NP-hard by a reduction from the undirected $s - t$ Hamiltonian path problem.

*Proof of Correctness.* Suppose there is a simple path of weight $-(n-1)$ from $i$ to $j$. Since each edge has weight $-1$, this path must contain $n-1$ edges, and so it corresponds to a Hamiltonian path in the original graph. Conversely, suppose that the original graph has a Hamiltonian path, say starting at $i$ and ending at $j$. This path leads to a path of weight $-1$ in $G$, which doesn't use the edges $(i, j), (j, i)$, and so it will be found when considering the edge $(i, j)$.

*Runtime Analysis.* Shortest Path is in P because we have a deterministic polynomial time algorithm that solves it. The problem becomes polynomial if we assume that $N$ does not contain any cycles of negative length. Thus, it has a polynomial-time algorithm which makes it NP-hard. However, it is no NP-complete because this is not a decision-making problem, which makes it not belong to NP.

---

You may assume the following problems are NP-hard for your reductions:

- **Undirected $s$-$t$ Hamiltonian Path:** Given an undirected graph $G = (V, E)$ and two vertices $s, t \in V$, is there a Hamiltonian path from $s$ to $t$ in $G$? (A Hamiltonian path is a path that passes every vertex).
- **3-SAT Problem:** Given a 3-CNF formula $\Phi$ (where each clause as *at most* 3 variables), is there an assignment to $\Phi$ that makes it true?

---

**Fun with Algorithms.** You are given a puzzle consists of an $m \times n$ grid of squares, where each square can be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions: (1) every row contains at least one stone, and (2) no column contains stones of both colors.

It is easy to see that for some initial configurations of stones, reaching this goal is impossible. We define the Puzzle problem as follows. Given an initial configuration of red and blue stones on an $m \times n$ grid of squares, determine whether or not the puzzle instance has a feasible solution.

Prove that the Puzzle problem is NP-complete. **(+10 points)**

**Solution.** *Algorithm.* We prove this puzzle is NP-hard by a reduction from 3-SAT problem. Let $\Phi$ be a 3CNF boolean formula with m variables and n clauses. The size of the board is $m \times n$ . The stone placed as follows, for all indices $i$ and $j$: If the variable $x_j$ appears in the ith clause, we place a blue stone at $(i, j)$. If the negated variable $x_j$ appears in the ith clause of $\Phi$, we place a red stone at $(i, j)$. Otherwise, we leave cell $(i, j)$ blank.

*Proof of Correctness.* We claim that this puzzle has a solution if and only if $\Phi$ is satisfiable. This claim immediately implies that solving the puzzle is NP-hard. If $C$ is satisfiable, then $\Phi$ is satisfiable also: Pick a satisfying assignment $x$ to $C$ and consider every wire of this circuit. Let $y$ be the assignment of these wires to variables in $\Phi$. By the above part, $C$ and $\Phi$ are logically equivalent and thus $y$ satisfies every clause in $\Phi$. Finally, since for the output wire we have a singleton clause z(which is equivalent to $C(x)$ which in turn is 1), the $\Phi(y) = 1$. This means $\Phi$ is satisfiable.

If $\Phi$ is satisfiable, then $C$ is satisfiable also. Pick a satisfying assignment $y$ to $\Phi$ and consider the variables assigned to the input wires. Let $x$ be the assignment of these input wires in $C$. By the above part, $C$ and $\Phi$ are logically equivalent and so every wire of circuit $C$ on the input $x$ will get the same value as the corresponding variable $y$ in $\Phi$. This in particular means that the output wire gets the value 1 on the input $x$ and thus $C(x) = 1$. This means $C$ is satisfiable.

*Runtime Analysis.* We show that if we have a poly-time algorithm for this puzzle the above reduction runs in poly-time. This is true because the size of $\Phi$ is at most a constant factor larger than size of the circuit and we create $\Phi$ in linear-time from $C$. Thus a poly-time algorithm for 3-SAT on $\Phi$ implies a poly-time algorithm for Circuit-SAT on $\Phi$.

This concludes the proof of NP-hardness of this puzzle. Given that we already proved this puzzle is also in NP, this means it is NP-complete.

---

Consider solving **at most one** of the following two challenge yourself problems.

**Challenge Yourself (I).** The goal of this question is to give a simple proof that there are decision problems that admit *no* algorithm at all (independent of the runtime of the algorithm).

Define $\Sigma^+$ as the set of all *binary* strings, i.e., $\Sigma^+ = \{0, 1, 00, 01, 10, 11, 000, 001, \ldots\}$. Observe that any decision problem $\Pi$ can be identified by a function $f_\Pi : \Sigma^+ \rightarrow \{0, 1\}$. Moreover, observe that any algorithm can be identified with a binary string in $\Sigma^+$. Use this to argue that "number" of algorithms is "much smaller" than "number" of decision problems and hence there should be some decision problems that cannot be solved by any algorithm.

*Hint:* Note that in the above argument you have to be careful when comparing "number" of algorithms and decision problems: after all, they are both infinity! Use the fact that *cardinality* of the set of real numbers $\mathbb{R}$ is larger than the cardinality of integer numbers $\mathbb{N}$ (if you have never seen the notion of cardinality of an infinite set before, you may want to skip this problem). **(+10 points)**

**Solution.** *Algorithm.*

*Proof of Correctness.*

*Runtime Analysis.*

---

**Challenge Yourself (II).** Recall that in the class, we focused on *decision* problems when defining NP. Solving a decision problem simply tells us whether a solution to our problem exists or not but it does not provide that solution when it exists. Concretely, let us consider the 3-SAT problem on an input formula $\Phi$. Solving 3-SAT on $\Phi$ would tell us whether $\Phi$ is satisfiable or not but will not give us a satisfying assignment when $\Phi$ is satisfiable. What if our goal is to actually find the satisfying formula when one exists? This is called a *search* problem.

It is easy to see that a search problem can only be "harder" than its decision variant, or in other words, if we have an algorithm for the search problem we will obtain an algorithm for the decision problem as well. Interestingly, the converse of this is also true for all NP problems and we will prove this in the context of the 3-SAT problem in this problem. In particular, we reduce the 3-SAT-SEARCH problem (the problem of finding a satisfying assignment to a 3-CNF formula) to the 3-SAT (decision) problem (the problem of deciding whether a 3-CNF formula has a satisfying assignment or not).

Suppose you are given, as a black-box, an algorithm $A$ for solving 3-SAT (decision) problem that runs in polynomial time. Use $A$ to design a poly-time algorithm that given a 3-CNF formula $\Phi$, either outputs $\Phi$ is not satisfiable or *finds* an assignment $x$ such that $\Phi(x) = True$. **(+10 points)**

**Solution.** *Algorithm.*

*Proof of Correctness.*

*Runtime Analysis.*

---