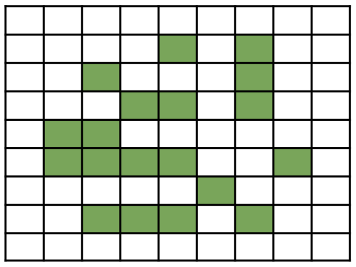
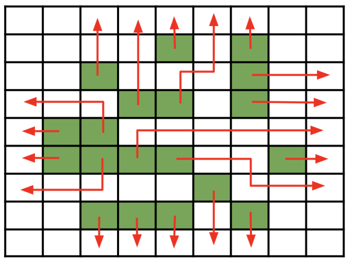
CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2021**

Homework #5: Solutions

May 5, 2021

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. See Figure 1:


(a) An "escapable" input


(b) A way of escaping the matrix

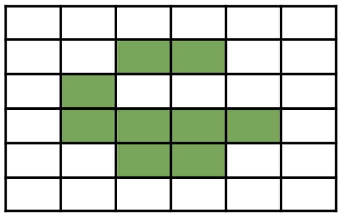

(c) A "non-escapable" input

Figure 1: The green cells correspond to the input k cells of the matrix.

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 (reduction to maximum flow problem):*

1. Create the following flow network $G = (V, E)$:
   - For every cell in the matrix, add a vertex to the graph $G = (V, E)$ and connect that vertex to all neighboring vertex-cells by bi-directed edges of capacity one.
   - Add a new vertex s and connect it to the k vertex-cells $(i_1, j_1), \ldots, (i_k, j_k)$ with edges of capacity one. Moreover, create a new vertex t and connect all boundary vertex-cells to t with edges of capacity one.
   - Assign a capacity of one to all *vertices* of this network other than s, t. This is done by using the technique of splitting each vertex into two, explained in Lecture 11 (see lecture notes).
2. Find the value of maximum s-t flow in this network and output the cells are escapable if and only if the max-flow value is k.

Proof of Correctness: By assigning capacity of one to all vertices and connecting s to k cells that needs to escape and connecting all boundary vertices to t, computing max-flow in this network is equivalent to finding maximum vertex disjoint paths from k cells to any boundary vertices. We prove that number of vertex-disjoint paths (i.e., value of flow) is k if and only if the cells are escapable.

(i) If the number of vertex disjoint paths (i.e., value of max-flow) is k then the cells are escapable: We can simply use the paths find by the algorithm to escape the cells since the paths are vertex disjoint and thus no cell will be used more than once. Moreover, since s is connected to k cells with edges of capacity one, the only way to have flow of k is to escape all the k input cells.

(ii) If the cells are escapable then the number of vertex disjoint paths (i.e., value of max-flow) is k: We define the vertex-disjoint paths (or the flow function) by picking each path that is used to escape the cells (namely, send one unit of flow over that path); since all k cells can escape this allows for a flow of size k which is feasible because no vertex will be shared in the paths. Also max-flow in this network is at most k since edges incident on s have capacity k in total and we clearly cannot send more flow.

Runtime Analysis: Number of edges in this network is $O(n^2)$ and the value of max-flow is at most k, hence using Ford-Fulkerson algorithm for max-flow will give a runtime of $O(n^2 \cdot k)$. Since $k = O(n^2)$ this runtime is $O(n^4)$ and *not* $O(n^3)$ asked in the question. However, notice that when $k > 4n$, the answer to this problem is trivially *No* (escaping is not possible): this is simply because we have $\leq 4n$ boundary cells in the matrix and we need one for each cell that escapes. As such, whenever $k > 4n$, we can simply output *No* in the algorithm and otherwise, when $k \leq 4n$, run the above algorithm to determine the answer. Thus, whenever we run the network flow algorithm, we have $k = O(n)$, which means the total runtime is $O(n^3)$.

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 (reduction from the network flow problem):*

1. Create a network $G' = (V', E')$ where $V' = L \cup R \cup \{s,t\}$. Connect $u \in L$ to $v \in R$ with an edge of capacity 1 whenever $\{u,v\}$ is an edge in the original graph. Moreover, connect s to every vertex in L with an edge of capacity p and every vertex in R to t with an edge of capacity q.
2. Compute a maximum flow f from s to t in the network G' and return its value as the answer.

Proof of Correctness: We prove that the size of maximum flow in G' is equal to the size of the largest (p,q)-factor in G. This is done by proving the following two assertions:

(*i*) If the maximum flow in G' is at least k, then there is a (p,q)-factor of size k.

Fix a maximum flow f in G'. Let M be the set of edges $\{u,v\} \in E$ with $u \in L$ and $v \in R$ such that $f(u,v) = 1$. We claim that M is a (p,q)-factor of size k. Firstly, since capacity of the incoming edge from s to any vertex $u \in L$ is exactly p, the flow incoming to u can be at most p and hence the flow going out of u can be at most p also: this means that the degree of u in M can be at most p. Similarly, using the fact that the capacity of the edge connecting each $v \in R$ to t is q, the flow going out of v, and hence the flow coming into v can be at most q also and hence the degree of v in M can be at most q. This means that M is a (p,q)-factor. Now since the value of flow f is k and all this flow needs to use the edges (u,v) with $u \in L$ and $v \in R$ to go from s to t, size of M is k also.

(*ii*) If the size of largest (p,q)-factor of G is k, then there is a flow of value of k in G'.

Fix a maximum size (p,q)-factor M in G. Define the flow f as follows: (1) for any $\{u,v\} \in M$ (with $u \in L$ and $v \in R$), let $f(u,v) = 1$, (2) for any $u \in L$, let $f(s,u)$ be equal to the number of edges incident on u in M, and for any $v \in R$, let $f(v,t)$ be equal to the number of edges incident on v in M. This is a feasible flow as the flow entering each vertex is equal to the flow out from that vertex, and since M is a (p,q)-factor, $f(s,u) \leq p$ for all $u \in L$ and $(v,t) \leq q$ for all $v \in R$, and thus capacity constraints are also satisfied. Moreover, the value of this flow is equal to the flow out of s which is equal to the degree of vertices in L inside M which is equal to the number of edges, i.e., k.

This implies that the maximum flow in G' is equal in size with the largest (p,q)-factor in G.

Runtime analysis: Constructing the network takes $O(n+m)$ time and running Ford-Fulkerson algorithm for maximum flow, takes $O(m' \cdot F)$ time where m' is the number of edges in G' and F is the value of maximum flow. Since $m' = m + 2n$ and $F \leq \min\{n \cdot p, n \cdot q\} \leq n \cdot (p+q)$, we obtain that the runtime of the algorithm is $O((n+m) \cdot n \cdot (p+q))$.

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 (**Solution 1**)**.** We show that a graph is 2-colorable if and only if it is bipartite. Then, we can use the algorithm from practice midterm 2, Problem 5, to solve this (in Solution 2 below, we have provided essentially the same algorithm as bipartiteness testing directly in the context of 2-coloring).

Algorithm (graph reduction): Run the algorithm of Problem 5 in practice midterm 2 to check if G is bipartite or not. Output G is 2-colorable if the algorithm says G is bipartite, and otherwise output G is not 2-colorable.

Proof of Correctness: As any other reduction, we prove the following two parts:

– If a graph G is bipartite, then it is 2-colorable.
  Because G is bipartite, it means that there are two sets of vertices L and R such that every edge of G is between L and R (no edge has both endpoints in L nor R). We can thus color L with one color and R with another, and have that no edge has the same color on both its endpoints. This means G is 2-colorable.

– If a graph G is 2-colorable, then it is bipartite.
  Because G is 2-colorable, it means that we can color the vertices with only two colors such that no edge has both its endpoints colored the same. We can thus define L to be the set of vertices that are colored with the first color, and R be the vertices colored with the second color. This way, all edges of G are going to be between L and R, meaning that G is bipartite.

This means that G is 2-colorable if and only if G is bipartite. Thus, the algorithm outputs the correct answer.

Runtime analysis: The runtime of the algorithm is the same as the one for bipartiteness testing which was $O(n+m)$ time. This is polynomial time.

Solution (**Solution 2**)**.** In this solution, we repeat the algorithm for bipartiteness testing in Problem 5 of practice midterm exam 2, but this time directly for the 2-coloring problem (note that in general, it is preferable to do a graph reduction, as in Solution 1 above, instead of modifying or designing a new algorithm for a given graph problem).

Algorithm: We assume G is connected; otherwise, we can run this algorithm on each connected component of G individually.

Algorithm: Pick an arbitrary vertex $s \in V$. Run BFS (for shortest path) from s and define the layers $L_1, L_2, \ldots,$ of vertices where

$$L_i := \{v \in V \mid d[v] = dist(s,v) = i\}.$$

For each vertex v, store 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 for some i, output G is not 2-colorable and terminate. Otherwise, if the for-loop never terminated, output G is 2-colorable.

Proof of Correctness:

- If G is 2-colorable, then the algorithm outputs 2-colorable. Assume toward a contradiction that the algorithm outputs G is not 2-colorable. 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 $P^u_j = P^v_j$ (i.e., the last time these two paths collided) and suppose w is the j-th vertex of these paths (the last shared vertex of the 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 2-colorable. Suppose w is colored with the first color, then the second vertex should be colored with the second color, the third with the first color, and similarly till the end. But because C has an odd length, this means that we color the last vertex before w (in the cycle) also with the first color. So the edge between w and this vertex has the same color on both endpoints, a contradiction. This means that G cannot be a 2-colorable graph if the algorithm outputs not 2-colorable.

- If the algorithm outputs 2-colorable, then G is also 2-colorable. Consider coloring s with the first color, L_1 with the second, L_2 with the first, L_3 with the second, and so on and so forth. We prove that no edge of G has received the same color on both endpoints.

  Since $L_1, L_2, \ldots,$ 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 vertices with different colors by the construction above. This implies that G has a 2-coloring.

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. The verifier asks for the following proof: provide a valid k-coloring of G where the colors are in the set $\{1,2,\ldots,k\}$. Clearly, if G is k-colorable, there is such a proof, and otherwise no proof exists.

To verify the given coloring, we check:

1. There is no edge (u,v) with both endpoints colored the same. This can be done using a for-loop over edges and checking the colors of both endpoints.
2. There is no vertex that is colored with any color other than $\{1,\ldots,k\}$. This can be done using a for-loop over vertices and checking its color.

This verifier algorithm then can output whether the given coloring is a valid k-coloring of G or not.

The runtime of the verifier is $O(m)$ for the first part and $O(n)$ for the second, so $O(n+m)$ in total. This 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. In the following, we say that a graph G has a 1/4-Path if there is a path in G that passes through *at least one forth* of the vertices in G.

Reduction (from Undirected s-t Hamiltonian Path):

(1) Given an instance $G = (V, E)$ of the undirected s-t Hamiltonian path problem, we create an instance G' of 1/4-Path as follows.

(2) Obtain G' by adding $3n + 7$ dummy vertices to G and connecting them to s and adding one additional dummy vertex and connecting it to t.

(3) We then run the algorithm for 1/4-Path on G' and output G has s-t Hamiltonian path if and only if the algorithm outputs G' has a 1/4-path.

Proof of correctness. We show that G has an s-t Hamiltonian path if and only if G' has a 1/4-path. Note that the number of vertices in G' is $4n + 8$, hence any valid path for the 1/4-Path problem in G' has to be of length at least $n + 2$.

(i) If G has a s-t Hamiltonian path P, then G' has a 1/4-path:
The path $P' = d_s \to s \rightsquigarrow_P t \to d_t$ (the notation $s \rightsquigarrow_P t$ is used to show we copy the path P from s to t here), where d_s is one of dummy vertices connected to s and d_t is the dummy vertex connected to t, is a path in G' that passes through exactly $n + 2$ vertices. Thus, it is a valid answer for the 1/4-Path problem.

(ii) If G' has a 1/4-path, then G has a s-t Hamiltonian path:
Note that any path of length more than 3 cannot contain two dummy vertices connected to s and hence the maximum length of a path in G' is $n+2$. Moreover, any path of length $n+2$ in G' must contain a dummy vertex connected to s, all original vertices (n vertices), and the dummy vertex connected to t and consequently has to start from and end in a dummy vertex. Hence if G' has a path of length $n + 2$, then removing the first and last (dummy) vertices results in a Hamiltonian path from s to t in G.

This implies the correctness of the reduction.

Runtime analysis: This reduction can be implemented in $O(n + m)$ time and hence a poly-time algorithm for 1/4-Path implies a poly-time algorithm for undirected s-t Hamiltonian path which in turn implies P = NP (by definition of s-t Hamiltonian path being NP-hard). Thus a poly-time algorithm for 1/4-Path also implies P = NP, making this problem NP-hard.

NP-completeness? Yes. 1/4-Path belongs to NP because we can have a *verifier* that takes such a path as a proof and simply check whether this is path that goes through more than one fourth of the vertices and all edges of the path belong to the graph, all in polynomial time. Since this problem is both in NP and also NP-hard, it is also NP-complete.

(b) **Two-Third 3-SAT Problem:** Given a 3-CNF formula Φ (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. We show that any instance Φ of 3-SAT problem can be solved in polynomial time if we are given a poly-time algorithm for two-third 3-SAT problem.

Reduction (given an input Φ of 3-SAT problem):

(1) Let m denote the number of clauses in Φ.

(2) We create a new 3-CNF formula Φ' by adding all variables and clauses of Φ, as well as defining m *new* variables $z_1, \ldots, z_m$ and adding clauses $(z_1), \ldots, (z_m)$ and $(\bar{z_1}), \ldots, (\bar{z_m})$ to Φ, i.e.,

$$\Phi' = \Phi \wedge (z_1) \wedge \cdots (z_m) \wedge (\bar{z_1}) \wedge \cdots (\bar{z_m}).$$

(3) We run the algorithm for two-third 3-SAT on Φ' and output Φ is satisfiable if and only if that algorithm outputs Φ' has an assignment that satisfies 2/3 of clauses.

Proof of Correctness: We prove that Φ is satisfiable if and only if Φ' has an assignment that satisfies 2/3 of its clauses.

(1) If Φ is satisfiable then Φ' has an assignment that satisfies 2/3 of clauses:
Let x be a satisfying assignment of Φ. Consider the assignment of x to Φ-part of Φ' and assigning True to all variables $z_1, \ldots, z_m$. Clearly Φ part and clauses $(z_1), \ldots, (z_m)$ are all satisfied in Φ' which constitute $2m$ clauses. Since the total number of clauses in Φ' is $3m$, this implies this assignment satisfies 2/3 of clauses of Φ', hence Φ' has such an assignment.

(2) If Φ' has an assignment that satisfies 2/3 of clauses then Φ is satisfiable:
Let y be an assignment that satisfies at least 2/3 of clauses in Φ'. Note that since we have both clauses (z_i) and $(\bar{z}_i)$, any assignment can satisfy exactly half the new clauses of Φ'. Hence, to for this assignment to satisfies 2/3 of total clauses, it should also satisfies all clauses of Φ in Φ'. This means that in this case Φ is satisfiable.

This implies the correctness of the reduction.

Runtime analysis: Constructing the new formula takes polynomial time in size of Φ as we are simply copying Φ and adding m new clauses in $O(m)$ time. Thus any polynomial time algorithm for two-third 3-SAT problem implies a polynomial time for 3-SAT which is an NP-hard problem (and thus it having a poly-time algorithm means P=NP). As such a poly-time algorithm for two-third 3-SAT problem implies P = NP and so two-third 3-SAT is also NP-hard.

NP-completeness? Yes. 2/3-SAT belongs to NP because we can have a *verifier* that takes such an assignment that satisfies 2/3 of clauses as a proof and simply check the number of clauses that are being satisfied by the assignment and accept the proof if there are at least 2/3 of clauses. This can be done in polynomial time. Since this problem is both in NP and also NP-hard, it is also 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. We show that any instance G, s, t of s-t Hamiltonian path problem can be solved in polynomial time if we are given a poly-time algorithm for negative-weight shortest path problem.

Reduction (given an input G, s, t of s-t Hamiltonian path problem):

(1) Assign a weight of -1 to every edge of G.

(2) Run the algorithm for the negative-weight shortest path problem on G, s, t with the new weights.

(3) If the weight of the path found by the algorithm is $-(n-1)$, output there is an s-t Hamiltonian path in G and otherwise output no such path.

Proof of Correctness: We prove that G has a Hamiltonian s-t path if and only if the shortest path from s to t in G has weight $-(n-1)$.

(1) If G has a Hamiltonian s-t path then the shortest path from s to t in G has weight $-(n-1)$:
Let P be a Hamiltonian path from s to t in G. This path has weight $-(n-1)$ under new weights as it passes through every vertex and thus has $n-1$ edge of weight -1. Moreover, any path in G has at most $n-1$ edges and thus no path can have weight less than $-(n-1)$. Hence, the shortest s-t path has weight $-(n-1)$ in this case.

(2) If the shortest path from s to t in G has weight $-(n-1)$ then G has a Hamiltonian s-t path:
Let P be the shortest s-t path in G with new weights. Since weight of P is $-(n-1)$ it has exactly $n-1$ edges and thus n vertices. But then it means that P is a Hamiltonian s-t path in G.

This implies the correctness of the reduction.

Runtime analysis: Constructing the new weights takes polynomial time in the size of the graph. Thus any polynomial time algorithm for negative-weight shortest path problem implies a polynomial time for s-t Hamiltonian path problem which is an NP-hard problem (and thus it having a poly-time algorithm means P=NP). As such a poly-time algorithm for the negative-weight shortest path problem implies P = NP and so negative-weight shortest path is also NP-hard.

NP-completeness? No, because this is *not* a decision problem and so it does 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 Φ (where each clause as *at most* 3 variables), is there an assignment to Φ that makes it true?