CSOR W4231 Analysis of Algorithms I - Spring 2023

Homework 5

Tong Wu, tw2906

Problem 1

Section i

The problem requests a necessary condition for a feasible circulation with demands to exist, which should be, said from source and sink, that the sum of demands from all source nodes S should equal to the sum of demands from all sink nodes V. It should be the equation below as the mathematic form:

$$\sum_{v \in V} f^{\text{in}}(v) = \sum_{v \in V} f^{\text{out}}(v)$$
$$\sum_{v \in V} f^{\text{in}}(v) - \sum_{v \in V} f^{\text{out}}(v) = 0$$

According to the demand constraint $f^{\text{in}}(v) - f^{\text{out}}(v) = d(v)$, the equation should be expressed to the following expression:

$$\sum_{v \in V} d(v) = \sum_{v \in V} (f^{\text{in}}(v) - f^{\text{out}}(v)) = 0$$
$$\begin{aligned}\sum_{v \in V} d(v) &= \sum_{d(v)>0} d(v) + \sum_{d(v)=0} d(v) + \sum_{d(v)<0} d(v) = 0 \\ &= \sum_{v \in T} d(v) + \underbrace{\sum_{d(v)=0} d(v)}_{=0} + \sum_{d \in S} d(v) = 0 \\ &= \sum_{v \in T} d(v) + \sum_{d \in S} d(v) = 0\end{aligned}$$

Hence, the necessary condition for a feasible circulation with demands to exist is:

$$\sum_{v \in T} d(v) = -\sum_{d \in S} d(v)$$

Section ii

The problem requests to reduce the problem to finding max flow. The original flow networks has a set of source nodes S and a set of sink nodes T. In order to find max flow, a new source node s and a new sink node t has been introduced to connected all source node and sink node. So, a new flow network graph $G' = (V', E')$ has been created

based on the origin graph G, where $V' = V \cup \{s, t\}, E' = E \cup \{v, t\}$. All source nodes in set S is connected to new source node s, and all sink nodes in set T is connected to new sink node t. For each node $V \in S$, the capacity of edge (s, v) is set to $|d_t|$, and for each node $V \in T$, the capacity of edge (v, t) is set to d_t. Hence, the max flow problem can be solved in new flow network graph G' where the maxflow should equal to $\sum_{v \in T} d(v)$.

Problem 2

Section a

Part i

Main Idea

In this problem, an edge need to be found which can reduce the maximum flow if it is deleted. In this case, the problem can be converted to finding the maximum flow. The main idea is below.

1. First use Ford-Fulkerson algorithm to find the maximum flow f and generate the residual graph G_f
2. Perform BFS algorithm to find the set of nodes N reachable from s in G_f
3. For each found edge, if the starting node of the edge is in N and the ending node is not, then this edge is a critical edge.

Pseudo Code

```
def Find_Critical_Edge(G, s, t):
    # Ford-Fulkerson
    f = FF(G, s, t)
    # Residual graph
    G_f = Residual_Graph(G, f)
    # Search nodes
    N = BFS(G_f, s)
    for (u, v) in E:
        if (u in N) && (v !in N):
            return (u, v)
        end if
    end for
```

Correctness

Base case:

The graph has 0 edges. For this case, there should be no critical edges since there has no edges and no edges to remove. The function will return NULL which proof its correctness.

Induction hypothesis:

Assume the algorithm works correctly for the graph with n-1 edges, and proof that the algorithm will work fine for the graph with n edges.

Induction:

If add a non-critical edge (x, y) to the graph, the max flow in the graph will not change. The residual graph will generate a backward edge (y, x) which has the same capacity with (x, y). Then, if node x is reachable from the set N, then the node y should also be reachable from the set N.

Otherwise, if a critical edge (x, y) is added to the graph, the max flow in the graph will change. There will have no backward edge (y, x) in the residual graph. Hence, if node x is reachable from the set N, then the node y will not be reachable from the set N. Therefore, the algorithm is correct.

Running Time

1. Call Ford-Fulkerson algorithm will cost $O(VEU)$ running time.
2. Generate residual graph and call BFS algorithm both of each will cost $\mathrm{O}(V+E)$
3. The for loop checking edge will cost $\mathrm{O}(E)$ time

Hence, the running time of this algorithm will cost $\mathrm{O}(VE)$

Part ii

Main Idea

In this problem, an edge set containing all critical edges is requested to be found. In this case, one thought is to solve this problem in finding the maximum match of bipartite graphs. The basic idea is similar with the previous part, but Hopcroft Karp algorithm will be used in order to get the efficient algorithm. The main idea is below:

1. Convert flow network into bipartite graph, add new node and set each edge with 1 capacity.
2. Use Hopcroft Karp to find the maximum match
3. Find minimum vertex cover in the bipartite graph
4. Return the critical edge set by mapping the critical edges in the original flow network graph

Pseudo Code

```
def Find_Critical_Edge_Set(G, s, t):
    G_b = transform_to_bipartite(G)
    m = hopcroft_krap(G_b)
        # Search nodes
    for (u, v) in m:
        if (u in N) && (v !in N):
            critical_edge.append(u, v)
        end if
    end for
    return critical_edge
```

Correctness

The algorithm first change the flow network graph into bipartite graph. Then use Hopcroft-Karp algorithm to find the maximum matching in bipartite. Finally use the similar process in last part to find all critical edges and store them into a set. The induction of Hopcroft-Karp algorithm should be similar with the last algorithm in part i.

Running Time

1. Convert flow network graph into bipartite graph cost $\mathrm{O}(E)$ time, where E is the number of edges in the flow network
2. The running time of Hopcroft-Karp algorithm is $\mathrm{O}(\sqrt{V}E)$ which is mentioned in the lecture.
3. The running time of finding all critical edges is $\mathrm{O}(E)$

Hence, the total running time of this algorithm should be $\mathrm{O}(\sqrt{V}E)$

Section b

Main Idea

In this problem, the critical edge is computed at wrong capacity which should be $c_{uv}-1$, but count as c_{uv}. The problem ask to find a optimal flow faster than recomputing the max flow. In the previous section, the running time of computing the max flow in a flow network graph is $\mathrm{O}(VE)$. In this case only one edge change the capacity, which means recomputing the max flow is not important, but need to adjust the current max flow. The main idea is to use BFS algorithm search the substitute edge starting from node u. The main idea is below:

1. Adjust the capacity of edge (u, v) from c_{uv} to $c_{uv}-1$
2. Perform BFS algorithm to find new max flow starting from node u
3. Determine if new path has different capacity than before. If yes, update the max flow of this graph

Pseudo Code

```
def update_max_flow(G, c, u, v, flow):
    # Update the flow on edge (u,v)
    flow[u][v] -= 1
    # Perform BFS
    new_path = bfs(G, c, u, v)
    flow = update_flow(flow, new_path)
    return new_path, flow
```

Running Time

1. Update the flow on edge (u,v) costs $\mathrm{O}(1)$ time
2. Using BFS to find the alternative path from node u to node v will cost $\mathrm{O}(E+V)$ time
3. When update the flow of the path, the running time should be $\mathrm{O}(p)$, where p is the length of path. Hence in the worst case, the path length should be V then the running time should be $\mathrm{O}(V)$.

Hence, the total running time of this algorithm is $\mathrm{O}(E+V)$

Problem 3

Section a

In this problem, an expected number of clauses satisfied by the algorithm is requested in terms of m, n, k_j, where m stands for the number of clauses, n stands for the total number of variables, k_j stands for the literals in j-th clauses.

For a specific clause j, it has k_j literals, and each literal is either 1 or 0. Since the process of determine the value is by flipping a coin, so the possibility for each literal is satisfied or not satisfied is both $\frac{1}{2}$. There are total

k_j literals, so the clause can be satisficed by at least one literal is satisfied has probability $1 - \frac{1}{2}^{k_j}$.

Hence, the equation can be written as:

$$E[X] = \sum_{j=1}^{m} (1 - \frac{1}{2}^{k_j})$$

The lower bound can be considered that the value of $k_j = 1$, hence the probability that each clause has been satisfied is $\frac{1}{2}$. Filled into the equation, the lower bound should be:

$$E[X] = \frac{1}{2}^{m}$$

Section b

This problem change the way of truth assignment. For the deterministic algorithm, each variable is assigned to a truth value, which can satisfies the maximum number of as-yet-satisfied clauses. For each loop that assign the truth value, there are at least $\frac{1}{2}$ of unsatisfied clauses will be satisfied, since at the worst case, the number of clauses that contain this variable with no “NOT” gate is equal to the clauses that has “NOT” gate.

As the result of the first loop, there are $\frac{1}{2}m$ clauses unsatisfied, this number changes to $\frac{1}{2}m + \frac{1}{2}^{2}m$ for the next loop. Hence, after n loops, the number of unsatisfied clauses has upper bound $\frac{1}{2}^{n}m$, the lower bound should be:

$$(1 - \frac{1}{2}^{n})m$$

Problem 4

Part a

MAX_SAT(D): Given a SAT expression ϕ which contains m clauses and n variables. Does a truth assignment exists that can satisfy at least T clauses, where $0 \leq T \leq m$?

Part b

In order to prove MAX_SAT(D) is NP-complete, need to first prove MAX_SAT(D) belongs to NP-class problem, then prove that MAX_SAT(D) is as hard as other NP-complete problem.

1. **Prove MAX_SAT(D) is NP-class problem:**

   For a Boolean variable truth assignment problem, it is able to check each clause is satisfied in polynomial running time. Hence MAX_SAT(D) is NP-class problem.

2. **Prove MAX_SAT(D) is as hard as other NP-complete problem**

   Reduce MAX_SAT(D) to 3SAT. 3-SAT problem has a input of CNF with n variables and each clauses has exactly 3 variables, and ask is there a truth assignment can satisfy the expression. In this case, the constructed instance is that, set the input of MAX_SAT as same as the input of 3-SAT. Then, set threshold T to the number of clause m. At this time, the decision problem of MAX_SAT is same as 3-SAT. If there is a solution for 3-SAT problem, then this solution is also satisfied to MAX_SAT. Hence, the reduction from MAX_SAT to 3-SAT is proved.

Therefore, since the MAX_SAT(D) is NP-class problem, and it is as hard as other NP-complete problem. Hence, the MAX_SAT(D) problem is NP-complete.

Part c

For brute-force algorithm, it needs to try all possible Boolean value, then find an assignment that can satisfy maximum number of clauses. There are total n variables and each variable can assign as 0 or 1, so there are 2^n possible assignments. For each assignment, it is important to check how many clauses satisfied. There are total m clauses so total running time to check all clauses is $\mathrm{O}(m)$. Hence the total running time is $\mathrm{O}(2^n m)$.

For the randomized algorithm in the problem, assume there are n values and assign n times. The total running time should be $\mathrm{O}(n)$. Which is a efficient algorithm comparing to the brute-force.

There is no contradiction between the results of part (a) and part (b). The two parts discuss different algorithms: part (a) analyzes randomized algorithm, while part (b) provides a lower bound on the number of satisfied clauses for a deterministic algorithm. Both algorithms have different performance guarantees, but part (a) shows that the deterministic algorithm has a better lower bound.