CSCI 270 Homework 9

Qilin Ye

March 22, 2023

1. Min Cut Polynomial-time Reduction

Proof of DECISION-OPTIMIZATION *equivalence.* Throughout this problem we fix G, s, t, and let DECISION(C) denote the result of "whether there exists a $s - t$ cut of capacity at most C" obtained from running DECISION.

DECISION $\leqslant_p$ OPTIMIZATION is trivial. An $s - t$ cut with capacity $\leqslant C$ exists if and only if the min capacity among all $s - t$ cuts $\leqslant C$. Therefore, using OPTIMIZATION as a black box, we set DECISION(C) to false if and only if $C <$ the output of OPTIMIZATION.

To show OPTIMIZATION $\leqslant_p$ DECISION, it suffices to find the smallest C with DECISION(C) = true. That is, we need to find C such that DECISION(C) = true but DECISION($C - 1$) = false. Note that once DECISION becomes true for some parameter, it remains true for bigger parameters.

We begin by defining C^- to be the sum of all non-positive (or just negative) edge capacities. By the definition, C^- servers as a lower bound for any cut on this graph. If DECISION(C^-) is true then we are immediately done, as C^- is the optimal value. Otherwise, DECISION(C^-) is false but similarly DECISION(C^+) is true where C^+ is the sum of all positive edge capacities. Using our previous observation, there exists an $C_0 \in [C^- + 1, C^+]$ where DECISION($C_0 - 1$) is false and DECISION(C_0) is true. To find it, we use binary search.

The runtime of computing C^- and C^+ requires $|E|$ additions and/or subtractions. The number of binary search iterations is linear with respect to the number of bits of $C^+ - C^-$ and does 3 arithmetic operations inside. The runtime is therefore weakly polynomial.

Algorithm 1: Solving OPTIMIZATION using DECISION

```
1  Inputs: graph $G$, source $s$, sink $t$, "black box" DECISION.
2  Initialization: $C^- = \sum_{e \in E : c_e \leqslant 0} c_e$ and $C^+ = \sum_{e \in E : c_e > 0} c_e$.
3  /* Run binary search on $[C^-, C^+]$ w.r.t. DECISION*/
4  if DECISION($C^-$) is true: return $C^-$
5  $L = C^-$, $R = C^+$, $M = (L + R)//2$
6  while $R - L > 1$ do
7      if DECISION($M$) is true then
8          $R = M$, $M = (M + L)//2$
9      else
10         $L = M$, $M = (M + R)//2$
11 return $R$
```

□

Proof of OPTIMIZATION-SEARCH *equivalence.* OPTIMIZATION $\leqslant_p$ SEARCH is trivial: if SEARCH outputs an optimal cut, we simply compute its capacity, and it will be the smallest capacity of all $s-t$ cuts.

Conversely, to show SEARCH $\leqslant_p$ OPTIMIZATION, we propose the following algorithm:

Algorithm 2: Solving SEARCH using OPTIMIZATION

```
1 Inputs: graph $G = (V, E)$, source $s$, sink $t$, black box OPTIMIZATION
2 Initialization: $S = \{s\}$
3 for each vertex $v \in V$ do
4     If there exists an edge $(s, v)$ then set capacity of $(s, v)$ to $\infty$
5     Else create an edge $(s, v)$ with capacity $\infty$
6     If OPTIMIZATION$(V, E)$ remains unchanged then $S \leftarrow S \cup \{v\}$
7     Revert all changes made to $E$
8 Output the cut $(S, S^c)$
```

For convenience, we say a vertex v belongs to a cut (S, S^c) if $v \in S$.

One immediate observation is that $v \in V$ is added to S if and only if v belongs to some $s-t$ minimum cut: suppose not, that v does not belong to any min cut, i.e., the newly added edge (s, v) is an outgoing edge for any min cut (which have capacity C). After the change, they will all have capacity ∞ and in particular the graph doesn't have any capacity C $s-t$ cut anymore. Conversely, if v is in some minimum $s-t$ cut, then the changes made to (s, v) will not affect the capacity of that cut, as all changes are made within the s-side. Therefore, the output of our proposed algorithm is the union of all min cuts on G. It remains to show that this is also a min cut. Since G is a finite graph, there are only finitely many cuts, it suffices by induction to prove that the union of two min cuts is also a min cut.

To this end, let $(S_1, S_1^c), (S_2, S_2^c)$ be two (different) $s-t$ min cuts. Define the following four sets:

$$A = S_1 \cap S_2, \qquad B = S_1 \backslash S_2 = S_1 \cap S_2^c, \qquad C = S_2 \backslash S_1 = S_1^c \cap S_2, \qquad D = S_1^c \cap S_2^c.$$

By construction the four sets partition V and $S_1 = A \cup B$ and $S_2 = A \cup C$. From definition of cuts,

$$c(S_1, S_1^c) = c(A \cup B, C \cup D) = c(A, C) + c(A, D) + c(B, C) + c(B, D) \tag{1}$$

and similarly

$$c(S_2, S_2^c) = c(A \cup C, B \cup D) = c(A, B) + c(A, D) + c(C, B) + c(C, D). \tag{2}$$

Note that$c(S, S^c) = -c(S^c, S)$, as an edge leaves S iff it enters S^c. In particular, $c(C, B) = -c(B, C)$.

Now consider two cuts where the s-side are given by $S_1 \cap S_2$ and $S_1 \cup S_2$, namely, $(A, B \cup C \cup D)$ and $(A \cup B \cup C, D)$. We have

$$\begin{aligned} c(A, A^c) + c(D^c, D) &= c(A, B) + c(A, C) + c(A, D) + c(A, D) + c(B, D) + c(C, D) \\ &= c(A, B) + c(A, C) + c(B, D) + c(C, D) + 2c(A, D) \\ &= c(A, B) + c(A, C) + c(B, D) + c(C, D) + 2c(A, D) + c(B, C) + c(C, B) \\ &= c(S_1, S_1^c) + c(S_2, S_2^c). \end{aligned}$$

Since (S_1, S_1^c) and (S_2, S_2^c) are $s-t$ min cuts, (A, A^c) and (D^c, D) must also be. In particular, $(S_1 \cup S_2, (S_1 \cup S_2)^c)$ is a min cut. This completes our main proof.

Below is my original proof which involves manipulations of edges as opposed to vertices. Although correct (I believe), the proof is significantly longer. I threfore decided to scrap and replace it with something slightly more readable, i.e., the one above.

Beginning of original proof

Conversely, to show SEARCH $\leqslant_p$ OPTIMIZATION, we (i) attempt to remove "irrelevant" edges until (ii) only edges corresponding to some optimal cut are left, and (iii) reconstruct S from these edges. For notation, we let OPTIMIZATION(V, E) denote the output of the OPTIMIZATION algorithm on $G = (V, E)$, source s, and sink t. We fix $C =$ OPTIMIZATION(V, E).

Consider an optimal cut with $c(S, S^c) = C$. Let $e = (u, v)$ be an arbitrary edge with $u, v \notin \{s, t\}$. Three possibilities:

(i) u, v are both in S or are both in S^c. In this case, $c(S, S^c)$ is completely unaffected by the removal of e, and correspondingly OPTIMIZATION$(V, E\backslash\{e\})$ = OPTIMIZATION$(V, E) = C$.

(ii) $u \in S, v \in S^c$ or the other way around. Removing edge e will change $c(S, S^c)$ as long as the capacity of edge e is nonzero. In some scenarios (but not always), this will lead to a different min-cut capacity, i.e., OPTIMIZATION$(V, E\backslash\{e\}) \neq$ OPTIMIZATION$(V, E) = C$.

The key observation is that (ii) is *necessary* (though not sufficient) for OPTIMIZATION to change after removing an edge. We therefore purpose the following polynomial-time reduction:

Algorithm 3: Solving SEARCH using OPTIMIZATION

```
Inputs: graph $G = (V, E)$, source $s$, sink $t$, "black box" OPTIMIZATION
Initialization: $C$ = OPTIMIZATION$(V, E)$, $G' = (V', E')$, copy of $G = (V, E)$, and $S = \{s\}$
While true:
    Find $e \in E'$ such that OPTIMIZATION$(V', E'\backslash\{e\}) = C$
    $E' \leftarrow E'\backslash\{e\}$.
    If no such $e$ exists then break
Run ModifiedBFS$(s, E', S)$ and return cut $(S, S^c)$.
Function ModifiedBFS(node $u$, edge set $\tilde{E}$, cut set $\tilde{S}$):
    $\tilde{S} \leftarrow \tilde{S} \cup \{u\}$
    for each neighbor $v$ of $u$ with $v \notin \tilde{S}$ do
        If $(v, u) \in \tilde{E}$ then return
        Else recursively call ModifiedBFS$(v, \tilde{E}, \tilde{S})$
```

For a brief correctness proof:

(1) Each iteration of the **while** loop removes an edge from E', so the **while** loop terminates after at most $|E|$ iterations. Similarly, The modified BFS further adds a constraint on the normal BFS so it also terminates in $\mathcal{O}(|V| + |E|)$ at most.

(2) Given a $s - t$ cut (S, S^c) on G, we defined the edges *associated* with (S, S^c) to be $\{e \in E : e$ leaves S or $S^c\}$. (When context is clear we drop "$s - t$" for convenience.)

LEMMA. After any iteration of the loop, there exists one cut (S, S^c) on G such that $c(S, S^c) = C$ and all of its associated edges are still contained in E'. Consequently, by the time the **while** loop terminates, the resulting E' still contains all edges associated with some cut (S, S^c) on G with $c(S, S^c) = C$.

Proof. This is easily seen by induction. $G = (V, E)$ certainly contains all edges associated with a cut (S, S^c) on itself, in particular the ones with capacities C.

Let $G' = (V', E') \subset G$ be given and assume by induction that OPTIMIZATION$(V', E') = C$. Let $(S_1, S_1^c), \ldots, (S_k, S_k^c)$ be the only cuts on G with capacities C whose associated edges are entirely contained in E'. If $e \in E'$ is associated with each cut and is removed, then $c(S_i, S_i^c)$ changes for each i. We have the two possibilities (if $c_e = 0$, removing it does not change anything):

If the capacity of e is positive, the after removing it we obtain a smaller optimal value, so OPTIMIZATION$(V', E'\backslash\{e\}) \leqslant$ OPTIMIZATION$(V', E') - c_e <$ OPTIMIZATION$(V', E') = C$.

On the other hand, if the capacity of e is negative, then each cut (S_i, S_i^c)'s capacity increases after removing e. On the other hand, all other cuts besides (S_i, S_i^c) had capacities $> C$ prior to removal of e. Therefore, after removing e, OPTIMIZATION$(V', E'\backslash\{e\}) >$ OPTIMIZATION$(V', E') = C$.

To sum up, in order to prevent OPTIMIZATION(V', E') from changing, each time after we remove an e, there must still be some cut (S, S^c) with capacity C such that its associated edges are entirely contained in E'. This proves the claim. $\square$

(3) Continuing on the result from the previous lemma, by the time the **while** loop terminates, all edges in E' must be associated with *one* cut whose capacity is C. To see this, suppose $c(S, S^c) = C$ and its associated edge set $E_S \subsetneq E'$. Pick $e \in E' \backslash E_S$. We can then remove e without affecting OPTIMIZATION(V', E') since E_S is still kept intact. This contradicts the termination assumption. Therefore, when the loop terminate, we obtain a set of edges that are precisely those associated with some capacity C $s-t$ cut.

(4) The modified BFS will then reconstruct the cut (S, S^c) whose associated edges are E'. After this, we are done.

(5) For runtime, (1) combined with the fact that each **while** loop takes at most $\mathcal{O}(|E|)$ work show that the total runtime is polynomial in $|E|$ and $|V|$. $\square$

End of original proof

2. NP Verifications

(a) Given a team assignment, it takes n/k additions to compute the sum of skills for each group, and so n total additions to obtain k numbers corresponding to the k skill sums of k groups. Traversing through these k numbers, keeping two pointers, one for minimum and one for maximum, it takes $\mathcal{O}(k)$ time to find the min and max, and finally, another subtraction to obtain Δ. All of these tougher take $\mathcal{O}(n)$ time. We call these actions COMPUTE_DIFF.

Form 1(a), it suffices to answer the decision question "given $C \in \mathbb{R}$, does there exist an assignment with $\Delta \leqslant C$?" For a given C, it is clear that the answer is YES if and only if there exists an input to COMPUTE_DIFF that outputs a $\Delta \leqslant C$. Furthermore, any valid assignment will be of polynomial length (when represented as binary strings) with respect to the given inputs. From the previous paragraph, this certification takes $\mathcal{O}(n)$ time.

(b) Given a set E, and each partition (S_i, S_i^c), a brute force solution for checking the number of edges cut by (S, S^c) is by simply iterating through all edges and maintain a counter. There are k partitions, so brute force iterating through all cuts takes $\mathcal{O}(k|E|)$ time to return the maximum number of edges cut by any one of the k partitions. We call these actions COMPUTE_EDGES_CUT.

Again, from 1(a), it suffices to answer the decision "given $z \in \mathbb{Z}^+$, does there exist a set E of edges with $|E| = z$ such that for each of the cuts (S_i, S_i^c), the number of edges it cuts in the graph (V, E) is at most C?" For a given z, it is clear that the answer is YES if and only if there exists a set E such that COMPUTE_EDUES_CUT outputs a number $\leqslant C$. Furthermore, a valid answer is just a cut, which is certainly polynomial size (represented as binary string) with respect to the graph. This verification process takes $\mathcal{O}(k|E|)$ time, and we are done.

3. SOKOBAN

(1) "Given a pattern of SOKOBAN (i.e., locations of walk-able squares, walls, an initial location, and a destination), does there exist a sequence of movements that moves the box from initial location to destination, using moves permitted by SOKOBAN?"

(2) A proposed sequence of SOKOBAN moves that moves the box from starting point to destination.

(3) An algorithm that checks (i) if the box starts at starting point, (ii) if the box ends at ending point, and (iii) if each movement in the proposed sequence is a valid SOKOBAN move.

(4) "**Efficient** certifier." We have not shown if this certifier runs in polynomial time. example, we have not ruled out the possibility that a correct solution, should it exist, happens to take at least exponential time of the input (i.e. the size of the board). Verifying such certificate would also take exponential time.

4. NP-completeness of CERTIFICATION

(1) It suffices to show CERTIFICATION has an efficient certifier. Let a certificate, i.e., a candidate quadruple $P, \hat{x}, \hat{t}, \hat{z}$ be given. Let y be a binary string of length $|y| \leqslant |\hat{z}|$ and we feed the input $(\hat{x}, y)$ into P and let the program run for at most $|\hat{t}|$ steps. It is clear that our CERTIFICATION is true if and only if for some y, this verification process terminates in $\leqslant |\hat{z}|$ steps and returns a YES. In particular, this process takes polynomial time w.r.t. $|\hat{t}|$, so CERTIFICATION is NP.

(2) Let X be any NP problem. By definition there exist polynomial functions p_1, p_2 such that the certifier B for X takes input x and y with $|y| \leqslant p_1(|x|)$ and outputs an answer in $\leqslant p_2(|x|)$ time. In particular, this certifier itself corresponds to the "P" in CERTIFICATION. That is, x corresponds to a "YES" for question X if and only if CERTIFICATION$(B, x, p_1(|x|), p_2(|x|))$ equals YES. This completes the polynomial-time reduction.