EECS 376: Foundations of Computer Science
University of Michigan, Winter 2024

Solutions to Homework 9

This homework has 8 questions, for a total of 100 points and 5 extra-credit points.
Unless otherwise stated, each question requires *clear*, *logically correct*, and *sufficient* justification to convince the reader.
For bonus/extra-credit questions, we will provide very limited guidance in office hours and on Piazza, and we do not guarantee anything about the difficulty of these questions.
We strongly encourage you to typeset your solutions in LaTeX.
If you collaborated with someone, you must state their name(s). You must *write your own solution* for all problems and *may not use any other student's write-up.*

(0 pts) 0. **Before you start; before you submit.**

If applicable, state the name(s) and uniqname(s) of your collaborator(s).

Solution:

(10 pts) 1. **Self assessment.**

Carefully read and understand the posted solutions to the previous homework. Identify one part for which your own solution has the most room for improvement (e.g., has unsound reasoning, doesn't show what was required, could be significantly clearer or better organized, etc.). Copy or screenshot this solution, then in a few sentences, explain what was deficient and how it could be fixed.

(Alternatively, if you think one of your solutions is significantly *better* than the posted one, copy it here and explain why you think it is better.)

If you didn't turn in the previous homework, then (1) state that you didn't turn it in, and (2) pick a problem that you think is particularly challenging from the previous homework, and explain the answer in your own words. You may reference the answer key, but your answer should be in your own words.

Solution:

(12 pts) 2. **Undecidable vs. NP-hard vs. NP-complete.**

Show that the undecidable language L_{ACC} is NP-hard, but is not NP-complete.

You may use the fact that $\mathsf{NP} \subseteq \mathsf{EXP}$ without proof. (This was the result of the extra-credit problem from HW7. Recall that EXP is the class of all languages that are decidable in exponential time, i.e., in time $O(2^{n^k})$ for some constant k where n is the input size.)

Solution: First, we prove that L_{ACC} is NP-hard by showing SAT $\leq_p L_{\text{ACC}}$. The reduction works as follows: given a formula φ, construct a TM M_φ that (ignores its input and) iterates through all the (finitely many) assignments of φ's variables. If some assignment satisfies φ, then M_φ accepts; otherwise it rejects (alternatively, it is also valid to loop). The reduction outputs the L_{ACC} instance $(M_\varphi, x = \varepsilon)$ (alternatively, any other string would be valid for x).

The reduction can construct M_φ in $O(|\varphi|)$ time, because the code of M_φ is basically just a loop with φ hard-coded into it. For correctness, observe that

- if $\varphi \in \text{SAT}$, then M_φ accepts all inputs, so $(M_\varphi, x) \in L_{\text{ACC}}$;
- if φ is not satisfiable, then M_φ rejects all inputs, so $(M_\varphi, x) \notin L_{\text{ACC}}$.

Next, we prove that L_{ACC} is not NP-complete, because it is not in NP. Since $\mathsf{NP} \subseteq \mathsf{EXP}$, every problem in NP is decidable in exponential time. So, since L_{ACC} is undecidable, it cannot be in NP.

(16 pts) 3. **Search vs. decision: any Hamiltonian path.**

Define the language

$$\textsc{AnyHamPath} = \{G : G \text{ is an undirected graph with a Hamiltonian path}\}\,.$$

Suppose that there exists an efficient algorithm D that decides $\textsc{AnyHamPath}$. Give an efficient algorithm that, on input an undirected graph G, outputs a Hamiltonian path in G if one exists, otherwise it outputs "No Hamiltonian path exists." Prove that your algorithm is correct and runs in polynomial time.

Solution: The general approach is as follows: firstly, we check if there exists a Hamiltonian path in G. We can do this by using the efficient algorithm D. If there isn't a Hamiltonian path in G, then we end. Otherwise, we go through each edge e of G, removing any edges that are not part of the Hamiltonian path. In the end, we use the remaining edges to form the Hamiltonian path. Below we present the algorithm to do this:

```
1: function $D'(G = (V, E))$
2:     If $D(G)$ rejects, return "No Hamiltonian path exists."
3:     for all $e \in E$: do
4:         Construct $G' = (V, E \setminus \{e\})$.
5:         If $D(G')$ accepts, let $G = G'$.
6:     Construct a Hamiltonian path $H$ in $G$ by finding a vertex $v$ of degree 1 and following
   edges starting from $v$
7:     return $H$
```

Runtime: D is efficient by assumption, and the size of G does not grow throughout the algorithm, so each call to D runs in time $O(n^c)$ for some constant $c > 0$, where $n = |G|$ is the (initial) size of the input graph. So, the loop runs in total time $O(n^c \cdot |E|) = O(n^{c+1})$ because it enumerates over the edges. Finally, constructing the output Hamiltonian path H

takes $O(n)$ time, because it scans through the vertices to find a suitable v, then follows $|V| - 1 \leq n$ edges. Therefore, the algorithm D' is efficient with respect to input size.

Correctness: If there is no Hamiltonian path in G, then the algorithm correctly outputs "No Hamiltonian path exists," because D is a decider for AnyHamPath. So, from now on, suppose that the input graph G has a Hamiltonian path. Below we show that when the algorithm halts, the only edges in G are *exactly* those of some Hamiltonian path H that is in the original input G, i.e., the final G is a "path graph." So, there are two vertices that have degree 1, and all others have degree 2, so following edges starting from either degree-1 vertex lists off the Hamiltonian path H (either in "forward" or "reverse" order).

We first prove that when the algorithm halts, there is a *unique* Hamiltonian path in G. First, there exists *at least one* Hamiltonian path in G, because each loop iteration preserves the existence of such a path. Now consider any two distinct Hamiltonian paths H_1, H_2 in the original input G; we claim that G cannot still contain both of these paths when the algorithm halts. Let e be the first edge the algorithm considers that is in one of H_1, H_2 but not the other. Then, when the algorithm considers deleting e from G, either some edge of $H_1 \cup H_2$ has already been removed (so the claim follows), or if not, one of the two paths is still in the modified graph $G' = (V, E \setminus \{e\})$. So, e would be removed, so both paths would not be in G when the algorithm halts.

Finally, we show that when the algorithm halts, there are no "extraneous" edges in G; that is, the edges of the unique Hamiltonian path H in G are the *only* edges in G. Consider any $e \notin H$. Then, when the algorithm considered deleting e from G, the modified graph would still have the Hamiltonian path H, so e would be deleted from G.

(20 pts) 4. **The fire-station problem.**

The state government is building fire stations in a region of Northern Michigan that has many small towns connected by roads. Since it's expensive to install a fire station in every town, the government has decided that it wants to build as few fire stations as possible, so that each town either has a fire station, or is directly connected by a road to a neighboring town that has a fire station.

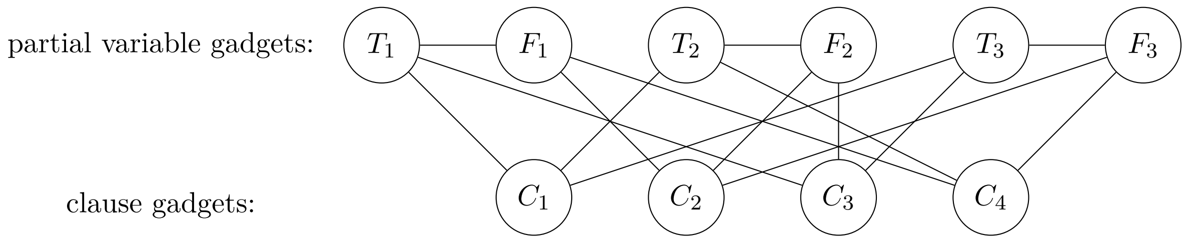
Formally, the decision problem FireStation is defined as follows: given an undirected graph $G = (V, E)$ and a positive integer "budget" k, does there exist a subset $S \subseteq V$ of at most k vertices such that for every $v \in V$, either $v \in S$ or there is an edge $(u, v) \in E$ such that $u \in S$? (Make sure to understand how this problem differs from the similar-looking VertexCover problem!) In this problem you will show that FireStation is NP-hard, by proving that 3SAT $\leq_p$ FireStation.

Since FireStation has some resemblance to VertexCover, a good source of inspiration is the 3SAT-to-VertexCover reduction from lecture. To start, we modify the reduction by making each "clause gadget" a *single vertex* (rather than a triangle of three vertices). To compensate for the simpler clause gadgets, the "variable gadgets" will need to be a little more complex, in a way that you will determine. In addition, the "budget" k will need to be different.

As an *incomplete* example of the modified reduction, consider the Boolean formula

$$\varphi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \ .$$

The reduction starts by building the following *partial* graph; you will define appropriate other nodes and edges for the variable gadgets:



Show the following two things:

1. that FireStation is NP-hard, by proving that 3SAT $\leq_p$ FireStation;
2. a diagram depicting the full output of your reduction for the formula φ given above.

As always, your proof will need to include all of the steps outlined in Handout 3: NP-Hardness Proofs.

Solution:

This problem is better known as DominatingSet. For an undirected graph G, a subset of vertices $S \subseteq V$ is a *dominating set* if for every vertex $v \in V$, either $v \in S$ or there exists a vertex $u \in S$ such that $(u, v) \in E$. (In other words, every vertex is within one edge of S.) We define the language DominatingSet $= \{(G, k) : G$ has a dominating set of size $\leq k\}$.

Our reduction is from 3SAT, and works as follows. Given a 3SAT instance (a 3CNF formula) ϕ, we output $(G, k = n)$ where the "budget" $k = n$ is the number of variables in ϕ, and graph G is constructed as follows.

- For each variable x_i in ϕ we create a "variable gadget" consisting of four vertices labeled T_i (for true), F_i (for false), α_i, and β_i, along with edges (T_i, F_i), (α_i, T_i), (α_i, F_i), (β_i, T_i), and (β_i, F_i).
- For each clause C_j in ϕ we create a single vertex labeled C_j, and introduce an edge between C_j and T_i if the literal x_i appears in C_j, and an edge between C_j and F_i if the literal $\overline{x_i}$ appears in C_j. So, there are up to three edges incident to each clause vertex C_j (there may not be exactly three because a clause may have a repeated literal).

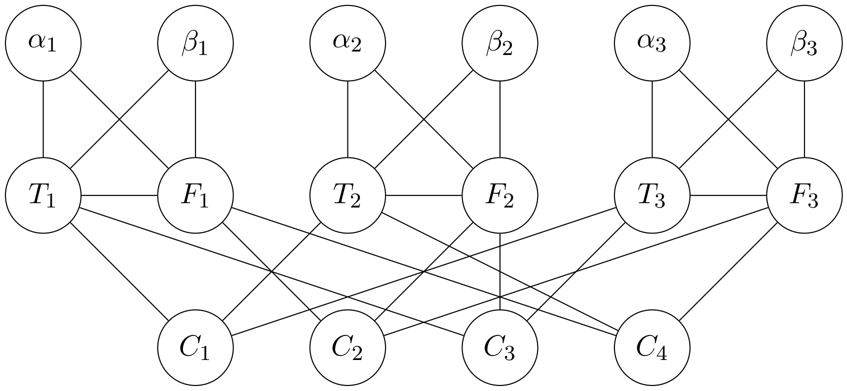
It is straightforward to see that this reduction takes polynomial time in the size of the input formula, because it creates $O(1)$ vertices and edges per vertex and clause.

(Note: compared to the partial reduction in the problem statement, we've added *two* vertices, α_i and β_i, to each variable gadget. As the proof below shows, we've done this to force a dominating set of size $\leq k = n$ to have exactly one of the T_i or F_i vertices in each gadget. However, the reduction also works if it adds just the single vertex α_i and its incident edges. In that case, however, a dominating set could potentially have α_i instead of T_i or F_i. Essentially, this means that the value of the corresponding variable doesn't matter, given the corresponding values of the other variables. This approach requires a bit more analysis in the analysis, but it can be proved correct.)

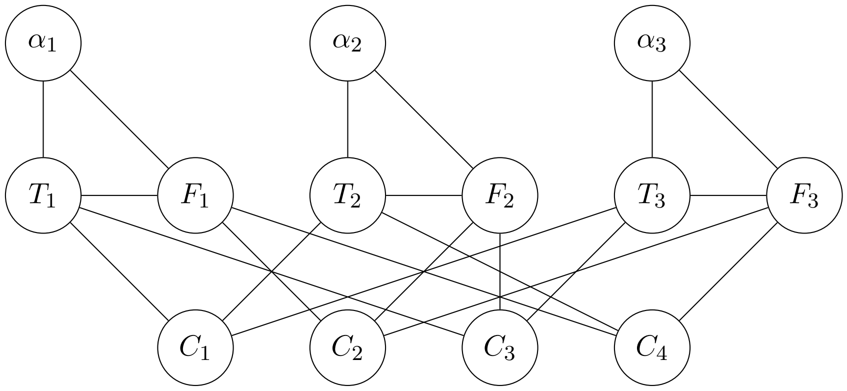
As an example of the reduction's output, for the above Boolean formula

$$\varphi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \,,$$

the graph is as follows:



For comparison, the output of the alternative reduction described above (parenthetically) would be:



We now show that the 3SAT instance is a "yes" instance if and only if the constructed instance of DOMINATINGSET is a "yes" instance, i.e,

$$\phi \in \text{3SAT} \iff (G, k) \in \text{DOMINATINGSET} \,.$$

For the forward direction, assume that $\phi \in$ 3SAT, i.e., there is a variable assignment that satisfies ϕ. We use such an assignment to exhibit a dominating set of size exactly $k = n$, by selecting exactly one of T_i or F_i for each variable gadget. Specifically, if x_i is assigned "true", select T_i, otherwise select F_i; so, the selected vertex is the one corresponding to the one true literal between $x_i, \overline{x_i}$. This results in a subset of exactly n vertices. The selected vertices form a dominating set because:

- for each variable gadget, all four of its vertices are "dominated" by selecting either its T_i or F_i vertex;
- because each clause C_j is satisfied by the assignment, it has at least one true literal, so the corresponding vertex C_j is a neighbor of at least one selected vertex.

Conversely, assume that our constructed DominatingSet instance $(G, k = n)$ has a dominating set S of size at most $k = n$. Notice that for each variable gadget, *at least one* of its four vertices must be in S, because the vertices α_i and β_i are not neighbors of any clause vertex. So, since S has size at most n, it must have *exactly one* vertex from each variable gadget. Moreover, since α_i and β_i are not neighbors, S must have exactly one of T_i or F_i. We use the selected vertices to define an assignment of the variables in ϕ, namely: if $T_i \in S$ then we set x_i to be true, otherwise (i.e., $F_i \in S$) we set x_i to be false; so, the assignment is such that the selected vertex corresponds to the one true literal between $x_i, \overline{x_i}$. We now show that this assignment satisfies all of the clauses in ϕ. Because S is a dominating set, each clause vertex has a neighbor in S, and since these neighbors correspond to the literals in the corresponding clause, at least one of the literals is true under the assignment, so the clause is satisfied. We conclude that the entire formula is satisfied by the assignment.

5. **Approximate knapsack.**

Recall the 0-1 *knapsack* problem, in which we are given a set of items having weights and values, and wish to select a subset of the items so that their total weight does not exceed a specified capacity, and their total value is maximized.

More specifically, an instance is a vector of weights $W = (W_1, \dots, W_n)$, a vector of values $V = (V_1, \dots, V_n)$, and a knapsack capacity C, all of which are non-negative integers. (Without loss of generality, each $W_i \le C$, because otherwise the ith item cannot be selected, so it is irrelevant.) The desired output is a subset $I \subseteq \{1, \dots, n\}$ of the items that maximizes the total value $\sum_{i \in I} V_i$, subject to the constraint that the total weight $\sum_{i \in I} W_i \le C$.

Recall from lecture that the Combined-Greedy algorithm for this problem is defined as follows:

1. Run the Single-Greedy algorithm,
2. Run the Relatively-Greedy algorithm,
3. Select one of the outputs that has the most total value (breaking a tie arbitrarily).

(Refer to the lecture for the definitions of the two sub-algorithms.)

In this question, you will prove that the Combined-Greedy algorithm is a 1/2-approximation algorithm for the 0-1 knapsack problem.

(8 pts) (a) Recall from lecture that one way to prove the correctness of an approximation algorithm for a maximization problem is to show a lower bound on ALG, the value obtained by the algorithm, and show a related upper bound on OPT, the value of an optimal fractional solution. In this part, you will show an upper bound on OPT.

To do this, it will help to recall from Homework 5 the *fractional* knapsack problem, a variant of the 0-1 problem that allows for taking any *partial* amount of an item. That is, for an item of weight W and value V, we may select some fractional amount $t \in [0, 1]$, which has weight $t \cdot W$ and value $t \cdot V$. Recall that the optimal value for 0-1 knapsack is at most the optimal value for the fractional version (on the same knapsack instance).

Prove that the optimal value OPT for the 0-1 knapsack problem is at most the *sum* of the values obtained by the two sub-algorithms.

Hint: It may help to recall from Homework 5 the greedy algorithm for the fractional knapsack problem, which is closely related to Relatively-Greedy (make sure to understand the difference!). You showed that this "Fractional-Greedy" algorithm obtains an optimal solution for fractional knapsack. You can use that fact here without repeating the proof.

Solution: Let OPT be the optimal 0-1 knapsack value, and S^* be the optimal fractional-knapsack value. Let S_1^* be the value obtained by the Relatively-Greedy algorithm, and S_2^* be the value obtained by the Single-Greedy algorithm. Because $\text{OPT} \le S^*$, it suffices to show that $S^* \le S_1^* + S_2^*$.

As stated in the hint, the optimal fractional knapsack value S^* is obtained by the Fractional-Greedy algorithm. So, $S^* \le S_1^* + F$, where $F \ge 0$ is the (fractional) value of the final (incomplete, fractional) item Fractional-Greedy considers—because before this point the Relatively-Greedy and Fractional-Greedy algorithms proceed identically. (After this point, Relatively-Greedy might select some other item(s), but this can only make S_1^* larger.)

Clearly, F is at most the full value of a most-valuable item, which is what the Single-Greedy algorithm obtains, i.e., $F \le S_2^*$. (Recall that any individual item can be taken in its entirety, because every $W_i \le C$.) Therefore, $S^* \le S_1^* + S_2^*$, as desired.

(6 pts) (b) Using the previous part, and by establishing a suitable lower bound on the value ALG obtained by Combined-Greedy, show that it is a 1/2-approximation algorithm for the 0-1 knapsack problem.

Solution: The "Combined-Greedy" algorithm achieves total value

$$\max\{S_1^*, S_2^*\} \ge (S_1^* + S_2^*)/2 \ ,$$

which by the previous part is at least $S^*/2 \ge \text{OPT}/2$. This proves the claim.

6. **Approximate f-SetCover.**

In this question, you will consider a variant of the SETCOVER problem where each element of the universe is in a limited number of subsets. Formally, in the f-SETCOVER problem, we are given a "universe" (set) U and subsets $S_1, \ldots, S_n \subseteq U$ *where each universe element appears in at most f of the subsets.* The goal is to find a smallest collection of the subsets that "covers" U, i.e., an $I \subseteq \{1, \ldots, n\}$ of minimum size such that $\bigcup_{i \in I} S_i = U$. We assume that $\bigcup_{i=1}^{n} S_i = U$, otherwise no solution exists.

You will analyze an approximation algorithm "f-cover" for the f-SETCOVER problem. The algorithm is a generalization of the "double cover" algorithm for VERTEXCOVER from lecture, and it works essentially as follows: while there is some uncovered element u in the universe, add to the cover *all* the subsets to which u belongs. The formal pseudocode is as follows. The notation $I(u) = \{i : u \in S_i\}$ for $u \in U$, that is, $I(u)$ indicates the subsets to which u belongs.

```
1: function f-COVER($U, S_1, \ldots, S_n$)
2:     $I = C = \emptyset$                                  ▷ selected indices $I$, covered elements $C$
3:     while $C \neq U$ do                                  ▷ not all elements are covered
4:         choose an arbitrary $u \in U \setminus C$        ▷ element $u$ is not yet covered
5:         $I = I \cup I(u)$, $C = C \cup \bigcup_{i \in I(u)} S_i$   ▷ add all subsets $S_i$ containing $u$ to the cover
6:     return $I$
```

Fix some arbitrary f-SETCOVER instance, and let I^* denote an optimal set cover for it. Let E denote the set of elements u chosen in Step 4 during an execution of the algorithm, and let I denote the algorithm's final output.

(7 pts) (a) Prove that $I(u) \cap I(u') = \emptyset$ for every distinct $u, u' \in E$. In other words, prove that if u and u' are each selected as the uncovered element in different iterations, none of the S_i contains both u and u'.

Solution: Consider two elements u, u' that are selected in line 4 in different iterations, and without loss of generality, suppose u was selected first. Then all the elements of $I(u)$ were added to I, making $I(u) \subseteq I$ for the rest of the algorithm. When we later select u', it must be that u' is uncovered by the subsets specified by I, so $I \cap I(u') = \emptyset$. (In other words, none of the subsets containing u' can be in the cover at the time we select u'.) Because $I(u) \subseteq I$, this implies that $I(u) \cap I(u') = \emptyset$, as desired.

(7 pts) (b) We want a lower bound on $\text{OPT} = |I^*|$. Using the previous part, prove that $|E| \leq |I^*|$.

Solution: By definition the subsets specified by I^* cover every element in U, and therefore every element in $E \subseteq U$. By the previous part, for each $i \in I^*$, the subset S_i can have *at most one* element of E in it. If not, we would have two distinct elements $u, u' \in E$ that are in S_i, meaning $i \in I(u) \cap I(u')$. So, we have an *injective* map from E to I^*, mapping each $u \in E$ to an $i \in I^*$ such that $u \in S_i$. We conclude that $|I^*| \geq |E|$.

(7 pts) (c) We want an upper bound on ALG $= |I|$. Prove that $|I| \leq f \cdot |E|$, and conclude that the f-cover algorithm is an f-approximation algorithm for the f-SetCover problem.

> **Solution:** For each $u \in E$, the algorithm adds at most f indices to I, because $|I(u)| \leq f$. So, $|I| \leq f \cdot |E|$. By part (b), we have that $|E| \leq |I^*|$, so $|I| \leq f \cdot |I^*|$, and therefore our algorithm outputs an f-approximation for f-SetCover.

(7 pts) (d) Prove that for every positive integer f, there is an input for which the f-cover algorithm necessarily outputs a cover that is *exactly* f times larger than an optimal one.

> **Solution:** Let f be an arbitrary positive integer. We construct an input on which the algorithm necessarily returns a cover that is exactly f times larger than the optimal cover size. Let the universe $U = \{1\}$, and define subsets $S_1 = S_2 = \cdots = S_f = U$. An optimal set cover is $\{S_1\}$, which has size 1. Our algorithm selects 1 in the first iteration, then returns the cover $\{S_1, \ldots, S_f\}$, which has size f. So for this example, the approximation ratio is exactly f.

(5 EC pts) 7. **Optional extra-credit question: disk storage**

You have been hired as a summer intern to work on a disk storage system. There are two identical disks, each with storage capacity of L. There are n files $F = \{f_1, \ldots, f_n\}$, where file f_i has size ℓ_i. A file cannot be split between the two disks, it must be stored entirely on one or the other (if it is stored at all).

The Disk Storage Optimization Problem is to store the *maximum number* of files from F on the two disks. It turns out that the decision version of this problem is NP-complete (maybe you can prove it!).

Give a polynomial-time algorithm that produces a solution—i.e., a selection of files and which disk to store each of them on—that stores a number of files *within just one* of optimal.

This result is neat, since all of the approximation algorithms we've seen so far come within some *multiplicative* factor of optimal. Here, the algorithm finds a solution that is only an *additive* amount worse than optimal!

> **Solution:** Consider the following algorithm. Iterate through the files in order from smallest to largest, putting as many of the files on the first disk as possible. When putting the next file on the first disk would exceed its capacity, start filling the second disk. Continue as before, putting as many of the remaining files as possible onto the second disk.
>
> Let S be the set of files obtained by going through F from shortest to longest file, and adding up the lengths of the files until right before the sum exceeds $2L$. $|S|$ is the largest number of files that can possibly fit onto the disks, so $OPT \leq |S|$.
>
> Now, we need to show that the algorithm finds a solution that fits at least $|S| - 1$ files onto the disks, that is, $ALG \geq |S| - 1$.

Suppose for contradiction, that $ALG \leq |S| - 2$. Then the solution found by the algorithm does not include the two longest files S_i and S_j of S (whose lengths are ℓ_i and ℓ_j respectively). Let u_1 and u_2 be the amount of unused space on disks 1 and 2, respectively. Since our algorithm greedily fills both disks, we know that ℓ_i and ℓ_j are both larger than both u_1 and u_2 (or else our algorithm would have included at least one of S_i or S_j). This means that $u_1 + u_2 < \ell_i + \ell_j$. But, by the definition of S, we know that $u_1 + u_2 \geq \ell_i + \ell_j$, a contradiction.

Thus, $ALG \geq |S| - 1$, completing the proof.

There are other greedy algorithms that also work, for example processing the files in order from smallest to largest and alternating which disk to put each file on.