University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn
S. Faour, M. Fuchs, G. Schmid

Algorithm Theory
Sample Solution Exercise Sheet 3

Due: Wednesday, 16th of November, 2022, 11:59 pm

Exercise 1: Quick Greedy Analysis *(7 Points)*

Given the following problems and proposed greedy strategies either formally prove or disprove (e.g. by counterexample) that the proposed greedy strategy is optimal for the given problem:

1. **Problem: Minimum Dominating Set** *(2 Points)*
   In an undirected graph $G = (V, E)$, with $n = |V|$.
   We want to find the smallest set S such that every node $v \in V$ either is in S, or has a neighbor in S (we say v is dominated by S).
   Or equivalently:

$$\forall v \ \in V : \ v \in S \lor \exists u \in S : (v, u) \in E$$

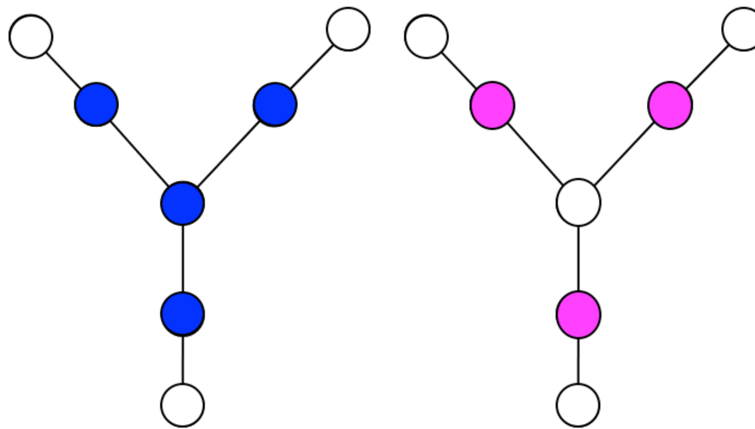
Strategy: Pick node that covers the most undominated neighbors
Nodes can be colored in 3 colors: *white*, *black* or *grey*. *White* represents nodes that are not yet dominated. *Black* represents nodes that are already part of our dominating set and *grey* represents nodes that are adjacent to a *black* node.
At the beginning all nodes are *white*. Then until no more *white* nodes exist, pick the node v^* that has the most *white* neighbors, or formally:

$$v^* = \arg\max_{v \in V} \deg_W(v)$$

where $\deg_W(v)$ is the number of *white* nodes adjacent to v.
Then color v^* *black* and all of the nodes adjacent to v^* *grey*. Repeat until no *white* nodes remain.
At the end all of the *black* nodes together form the Dominating Set.

Sample Solution

Figure 1a shows the greedy solution with a resulting dominating set of size 4. Fig 1b shows the corresponding optimal solution of size 3, therefore proving, that this greedy algorithm does not always output optimal solutions. Note that the greedy solution is not necessarily deterministic here, but it will always pick the middle node first and therefore never output an optimal solution.



(a) Greedy Solution on the given graph. The size of the Dominating Set is 4.

(b) Optimal Solution for the same graph. The size of the Optimal Solution is 3.

Figure 1: Shows a counter-example graph with greedy solution in blue and optimal solution in purple.

2. **Problem: Most finished jobs** *(3 Points)*
   We want to schedule n jobs $a_1, a_2, \dots, a_n$ with durations $d_1, d_2, \dots, d_n$. A job a_i is considered finished at time t if its starting time s_i fulfills the condition $s_i + d_i \leq t$.
   At each point in time there can be only one job currently worked on. We define this condition formally as: compute $s_1, s_2, ..., s_n$ such that they satisfy

$$\forall i, j \in \{1, \dots, n\}, j \neq i : s_j \notin \{s_i, s_i + 1, \dots, s_i + d_i\}$$

   or in words, that no job starts while another job is currently being worked on.
   Our goal is to compute $s_1, s_2, ..., s_n$ such that at every point in time t the number of already finished jobs is maximized.

   Strategy: Pick shortest job
   Whenever possible, meaning that whenever there is no job currently being worked on, pick the job with the shortest duration, that is not yet finished.

Sample Solution

Claim: Picking the shortest job first will produce an optimal schedule.
Note: We could have done this a bit quicker by a simple proof by contradiction, which follows the exact same steps, but i wanted to make an exchange argument as it is more in the spirit of greedy analysis.
Proof: We do a simple exchange argument, so assume there exists some schedule OPT that performs better than the schedule ALG that our greedy strategy will produce. We will describe a simple exchange step that will transform an optimal schedule into a greedy schedule without decreasing the quality of the solution.
W.l.o.G. we won't distinguish between jobs with the same duration. This means, that if two solutions only change the order in which jobs of the same duration are executed we consider them to be the same solution. This is valid, since swapping the position of these two jobs will not change the value of the solution.
If the optimal schedule is equal to the schedule our algorithm produces we are done and our claim is proven. So assume this is not the case, then there must be a smallest i such that the i-th job o_i in the optimal solution is different from the i-th job g_i in the algorithms solution. Since both solutions have processed the same jobs so far the optimal solution must process g_i at

some later time $\exists j > i : o_j = g_i$ we will call this timeslot o_j. We now simply change our optimal solution by swapping o_i with o_j. Since g_i is the job with the smallest duration among remaining jobs when it was chosen and $o_i \neq g_i$ we have that $d_{g_i} < d_{o_i}$ (if they were equal we wouldn't distinguish between them).
Therefore we will finish g_i faster than we would have finished o_i. As a result for all of the points in time between o_i and o_j , we will have that the number of finished jobs is less or equal to the number of jobs that were finished in the original solution.
After at most n such exchange steps we will arrive at the greedy solution. Hence we will always be able to transform our optimal solution into a greedy solution without making the solution worse, it follows directly, that the greedy solution has the same value as the optimal solution. This finishes the proof for our claim.

3. **Problem: Max-Cut** *(2 Points)*
In an undirected graph $G = (V, E)$, with $n = |V|$.
We define a cut $S \subset V$ to be a separation of V into two distinct subsets S and $V \backslash S$. We define the weight of a cut $w(S) = |\{(u, v) \mid u \in S, v \in V \backslash S\}|$ to be the number of edges between the two sets S and $V \backslash S$.
The goal is to find a maximum weight cut.

Strategy: Pick node that increases the cut the most
We define the relative-degree of $v \in V$ with respect to a subset $A \subset V$ to be

$$d_A(v) = |\{u \in A \mid (v, u) \in E\}|$$

being the number of edges from v to nodes in A.

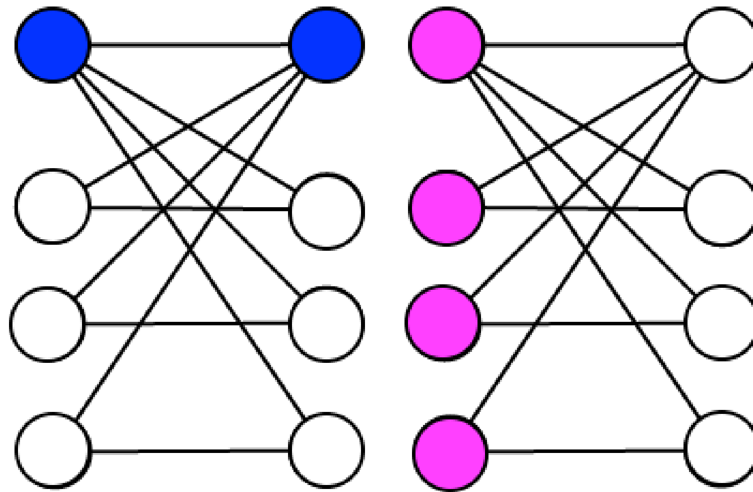
Again we start with an empty set $S = \emptyset$ and in every step try to get the biggest increase to our cut-weight. For this we either add a node v^* to our set $S = S \cup \{v^*\}$, or if already $v^* \in S$ we remove it from $S = S \backslash \{v^*\}$, where we determine v^* by the following characteristic:

$$v^* = \arg\max_{v \in V} \left\{ \begin{array}{ll} d_S(v) - d_{V \backslash S}(v), & \text{for } v \in S \\ d_{V \backslash S}(v) - d_S(v), & \text{for } v \in V \backslash S \end{array} \right\}$$

In words, we pick a v^* such that the increase to the cut would be biggest. Hence every such step will give us an improvement to our cut weight.
We stop once no $v \in V \backslash S$ can give us an improvement.

Sample Solution

Figure 2a shows the greedy solution with a resulting cut of size 6. Fig 2b shows the corresponding optimal solution resulting in a cut of size 10, therefore proving, that this greedy algorithm does not always output optimal solutions. Note that the greedy solution is deterministic here and will always pick the two blue nodes at the top. Afterwards it will terminate, since no node can give an improvement anymore.



(a) Greedy Solution on the given graph cutting 4.

(b) Optimal Solution for the same graph cutting 10 edges.

Figure 2: Shows a counter-example graph with greedy solution in blue and optimal solution in purple.

Exercise 2: Greedy Satisfying *(13 Points)*

We look at a boolean formula ϕ of n clauses in 3-Conjunctive Normal Form (CNF), meaning that it is given as a "logical and":$\wedge$ of clauses that only use "logical or's":$\vee$. In each clause we have exactly 3 literals, so an example would be:

$$\phi = (a \vee \bar{b} \vee c) \wedge (\bar{c} \vee d \vee e) \wedge (\bar{a} \vee \bar{d} \vee \bar{b})$$

Note: $\bar{a}$ stands for the negation of the variable a.

Devise an algorithm based on a greedy strategy to satisfy as many clauses as possible. Give an explanation why your algorithm is a greedy algorithm.
Make an effort to make the algorithm efficient! Argue the runtime of your algorithm, the runtime of your algorithm must be at most polynomial in n. You will most likely not be able to devise an algorithm that can satisfy all clauses/ that is optimal, since this problem is NP-hard. Therefor give an algorithm that is "as good as possible".

Prove a performance guarantee for your algorithm. More specifically this means, that you should prove that your algorithm can not be arbitrarily bad. Give a bound on $\frac{1}{2} < \alpha \leq 1$, such that your algorithm satisfies at least an α-fraction of clauses.

Hint: For the performance analysis, it might be helpful to think about the number of literals that have not been assigned thus far and look at how this number changes after assigning a literal.

Sample Solution

Some terminology first: A literal l is a variable in a phase, so for some variable a, $l \in \{a, \bar{a}\}$. As a direct consequence: $l = \bar{a} \Rightarrow \bar{l} = a$

Algorithm: Go through the variables in an arbitrary order and assign the literal that satisfies the most clauses. Then remove all of the satisfied clauses and all literals that are set to false.
E.g. say we check the variable a and assign its negation $\bar{a} =$ true. We will then delete all of the clauses that contain $\bar{a}$ and all of the occurrences of a in all of the remaining clauses.

Performance guarantee:

Claim: The greedy strategy will satisfy at least $\frac{3}{4}$ of clauses.
Proof: We use a potential argument, using the number of literals left in the clause. At the beginning of the algorithm the formula will contain $3n$ literals. As long as there are still literals left in the formula that we can assign, we can satisfy more clauses.
If we are able to show, that for each k clauses we satisfy we delete at most $4k$ literals, then it follows that we will satisfy at least $\frac{3}{4}$ of all clauses. We show this by induction over the number of literals $m = 3n$, we see that we can always satisfy $\frac{m}{4}$ clauses:

Base Case: For $m = 1$ simply assign this literal and we will have satisfied $1 > \frac{1}{4}$ of clauses.
Induction Hypothesis: Now assume that $\forall a \leq m$ literals we can satisfy $\frac{a}{4}$ of the clauses.
Induction Step: If we have $m+1$ literals, use the fact that we satisfy k clauses while deleting at most $4k$ literals, then we have that the number of satisfied clauses is at least $k + \frac{1}{4} \cdot (m + 1 - 4k) = \frac{1}{4}(m+1)$. Here we used that $m + 1 - 4k \leq m$ and we can therefore use the induction hypothesis.

We finish the proof by showing the following claim:

Claim: In each greedy step, if k clauses are satisfied, at most $4k$ literals are deleted.
Proof: Let l be the literal that we assigned to true. What we delete are all clauses which are now newly satisfied and all of the occurrences of the negation of our assignment $\bar{l}$. Since there are k clauses that are satisfied and each clause contains at most three literals, this will result in at most $3k$ literals being deleted. Since we assigned l and not $\bar{l}$ it must hold true that l occurs in more clauses than $\bar{l}$, therefore there are at most k occurrences of $\bar{l}$. So in total we delete no more than $4k$ literals, concluding the proof.

Runtime:

Claim: The greedy strategy can be implemented in $O(n)$ time.
Proof: We first build up a datastructure that will allow us to implement the greedy algorithm efficiently. We build up an array A indexed by our literals, at $A[i]$ is a List with references to all clauses in which literal i occurs. We can build this datastructure in time $O(n)$ by simply iterating over all of the clauses, notice that for each clause we have to add it to the end of three Lists, which takes constant time. And since the formula can contain at most $3n$ different literals the size of A is also in $O(n)$, by the same argument the number of references to clauses is $3n$. At the same time we build another array N in which we just save how often each literal occures in the formula. Note that this is simply three additional increment operations for each clause.
Now for each greedy step, say we are currently looking at variable b, we then have to check how often $\bar{b}$ and b occur in our formula. This is simply given by the values of our array $N[\bar{b}]$ and $N[b]$ respectively. So it takes constant time.
This leaves deleting the clauses and deleting the occurrences of the negated literal. However we don't actually need to delete anything, it suffices to adjust the values in N, since these will contain all of the information relevant for the rest of the algorithm. So we iterate over all of the satisfied clauses and for each literal l in each clause, we decrement the value $N[l]$. As a result after every step for every literal l, $N[l]$ will correspond to the number of clauses that will be newly satisfied if we assign $l = \text{true}$. Since we will never decrement a value below 0, the number of decrements over all greedy steps is upper bounded by $3n \in O(n)$.
Since the precomputation is $O(n)$, all n decisions are made by comparing just two values and decrementing is at most $3n$ decrements, we have a total runtime of $O(n)$. Thus proving our claim.