RWTH Aachen **WS 2018**
Lehrgebiet Theoretische Informatik Problem Set 2
Dreier—Kuinke—Rossmanith 29.10.2018

Parameterized Algorithms Tutorial

Tutorial Exercise T1

In this exercise, we wish to design an algorithm for the FEEDBACK VERTEX SET problem. An input to this problem is a graph $G = (V, E)$ and an integer k and the question is whether the graph has a vertex subset of size at most k whose deletion results in a forest. Such a vertex set is called a *feedback vertex set.*

1. A feedback vertex set hits all cycles in the graph. Therefore vertices which do *not* belong to cycles may be safely removed. Use this observation to design *reduction rules* to simplify the graph.
2. Can you bound the girth of the reduced graph? Use the bound to design an algorithm for the problem.
3. What is the running time of your algorithm?

Solution

We start by designing so-called *reduction rules.* These are polynomial-time algorithms that transform an instance (G, k) into an equivalent instance (G', k') such that the new instance is "simpler" or has "more structure." In this case, "equivalence" of input instances means: (G, k) is a yes-instance if and only if (G', k') is a yes-instance.

Reduction Rule 1**.** Iteratively delete vertices of degree one and keep the parameter fixed. Intuitively, this rule deletes "tree-like" portions of the graph and since no such vertex can be part of an optimal solution, this rule is sound.

Reduction Rule 2**.** Short-circuit vertices of degree two and keep the parameter fixed. That is, if v is a vertex of degree two with neighbors x and y (not necessarily distinct), then delete v and add an edge between x and y (even if they were connected by an edge). A vertex of degree two may or may not be in the solution, but any solution that selects this vertex can be transformed into a solution of the same size by removing it and picking one of its neighbors. This shows that this rule is sound. However this rule transforms a simple graph into a multi-graph. Try this rule on a cycle to see its effect.

ReductionRule 3**.** If a vertex has a loop then select it in the solution, delete it from the graph, and decrement the parameter by one.

Apply these rules in the order stated until they can't be applied any more. The instance that we obtain is said to be *reduced* wrt to the above rules. Any vertex in the reduced graph has degree at least three. An interesting property of such graph is that they always have a cycle of length at most $2 \log n + 1$. This can be seen by doing a BFS from an arbitrary vertex. Each vertex spawns two child nodes (except the root which spawns three children) and hence the depth of the BFS tree cannot exceed $\log n$. Hence there is a cycle of length at most $2\log n + 1$ (climb down at most $\log n$ edges in the tree, use a cross edge and then climb up at most $\log n$ edges).

Since one of the vertices in this cycle must be in an optimal solution (in particular, in a feedback vertex set of size at most k), we could branch on the vertices of the cycle. This gives us an algorithm with run-time $O((\log n)^k \cdot n^{O(1)})$. Another interesting fact is that $(\log n)^k$ is indeed an FPT-function (that is, it is bounded above by a function of the form $f(k) \cdot n^{O(1)}$).

Case 1. $k \leq \log n / \log\log n$. In this case, we have: $k \log\log n \leq \log n$. Raising both sides to the power of 2, we obtain:

$$(\log n)^k \leq n. \tag{1}$$

Case 2. $\log n / \log\log n < k$. In this case, we claim that $\log\log n < k$. Suppose not. Then we must have:

$$\log n < k \cdot \log\log n \leq (\log\log n)^2,$$

which is false. Hence $\log n < k^2$, which gives us:

$$n < 2^{k^2}. \tag{2}$$

By Inequalities 1 and 2, we conclude that

$$(\log n)^k \leq n + 2^{k^2}.$$

Tutorial Exercise T2

Given a boolean formula φ in CNF with n variables and m clauses and an integer k, you have to decide whether there exists an assignment to the variables that satisfies at least k clauses. Assume that the literals appearing in a clause are all distinct and that no clause contains a literal and its negation. We first consider several special cases.

1. If φ contains only unit clauses (clauses with one literal), then how can you find the optimum assignment?
2. If there are k clauses in φ that each contain k literals, then show that one can find an assignment satisfying all these clauses in $|\varphi|$ time.
3. Show that one can always find an assignment that satisfies at least $m/2$ clauses of φ.

Use these facts to design an FPT-algorithm with k as parameter.

Solution

1. Set a literal to `true` or `false` depending on which satisfies more clauses.
2. Let the clauses be denoted $C_1, \ldots, C_k$. Pick an arbitrary literal from C_1 and set it such that C_1 is satisfied. Inductively, proceed to C_i and pick a literal l_i that has not been set in the previous $i-1$ rounds. We are guaranteed that such a literal exists since there are at least k distinct literals in the k clauses that we started out with. Set l_i such that C_i is satisfied. The total time taken is linear in the sizes of all the clauses $C_1, \ldots, C_k$.

3. Start will the all-`false` assignment. If this does not satisfy at least half the clauses, then it falsifies half the clauses. But then the all-`true` assignment satisfies all those clauses falsified by the all-`false` assignment. In either case, you have an assignment that satisfies at least half the clauses.

Our algorithm works as follows:

1. If $k > m$ output "No" and halt.
2. If $k \leq m/2$ output "Yes" and halt.
3. Separate the clauses of φ into short and long clauses: short clauses have fewer than k literals and long clauses have at least k literals. Let φ_l and φ_s be the conjunction of the long and short clauses respectively. Let there be b long clauses. If $b \geq k$ then output "Yes" and halt.
4. Construct a binary tree of the following type: the root is labeled with the pair $(\varphi_s, k-b)$. In general, each node of the tree is labeled by a pair (ψ, j), where ψ is a boolean formula in CNF and j is a non-negative integer. If the label of a node satisfies one of these three categories it is a *leaf* node:
   (a) If j exceeds the number of clauses in ψ, then (ψ, j) is a leaf-node labeled "No."
   (b) If $j = 0$, then (ψ, j) is a leaf-node labeled "Yes."
   (c) If no literal in ψ occurs positively *and* negatively then (ψ, j) is a leaf node labeled "Yes."

   Pick a literal v that occurs both positively and negatively in ψ. Let the number of clauses that contain this literal in the positive form be l_{pos} and the number of clauses that contain it negatively be l_{neg}. Let ψ_v and $\psi_{\bar{v}}$ be the formulas obtained by setting v to `true` and `false`, respectively. Then (ψ, j) has two children labeled $(\psi_v, j - l_{\text{pos}})$ and $(\psi_{\bar{v}}, j - l_{\text{neg}})$. If the tree has a leaf node labeled "Yes", output "Yes" and halt. Since l_{pos} and l_{neg} are both at least 1 and j is at most k, we get a branching algorithm of 2^k.

Homework H1

Consider the following algorithm for VERTEX COVER. Choose an arbitrary edge $e = \{u, v\}$ that has not yet been covered and branch on the two subcases: on one branch include u in the solution and in the other include v in the solution. Return the smaller of the two solutions.

Does this algorithm run in FPT-time if parameterized by the size of the *minimum* vertex cover? If yes, provide a formal proof. If no, provide a generic counterexample.

Solution

This algorithm does not run in FPT-time. Consider the star graph S_r. The recursion tree of this algorithm on this graph will be one long path of length r with a single further vertex attach to each vertex of the path, in total this tree has $r + 1$ leaves. Now consider a graph consisting of l distjoint copies of S_r labeled $S_r^1, S_r^2, \ldots, S_r^l$. Without loss of generality, we assume that the algorithm first branches on the edges of S_r^1, then on the edges of S_r^2 and so on.

This results in a recursion tree of size $(r+1)^l$: regard the whole search tree for a single star as a node with $r+1$ children, then the depth of this reduced tree is exactly l.

The graph clearly has a minimum vertex cover of size l, but the size of the search tree and therefore the running time depends also exponentially on l. It follows that this algorithm does not run in FPT-time.

Homework H2

Consider the following algorithm for Independent Set. Given a graph $G = (V, E)$, we first check whether the maximum degree is at most two. If this is the case, we can find a maximum independent set (in polynomial time) and return the solution. Else we find a vertex u of largest degree (which is at least 3) and branch on it. There are two cases: we can either include u in the independent set in which case we must delete all its neighbors; if we choose *not* to include it in the independent set then we can safely delete it from the graph. We branch on these two cases and return the bigger of the two solutions.

1. Does this algorithm compute a largest independent set? How would you prove it?
2. Write down the recurrence that governs the running time of this algorithm. Solve the recurrence using methods from the lecture. What is the exponential factor in the running time?

In case you're worried about what the parameter is in this case, it is simply the number of vertices in the graph. Remember that the parameter *must* decrease in each branch of your recursion tree, and with this choice of parameter, this requirement is satisfied.

Solution

1.

This algorithm computes a largest independent set, because it iterates over all possible solutions. For every vertex we check if it is inside or not and by construction the Algorithm will never output a set S that is not an independent set.

Let S^* be an optimal independent set of G. We now show that our agorithm can find it. If G has bounded degree by 2 we are done. Otherwise the algorithm finds a vertex u. If $u \in S^*$, we consider the branch where the algorithm includes u in the set S. None of the neighbors of u are in S^* so deleting them is safe. If $u \notin S^*$ we consider the branch where we the algorithm does not include $u \in S$. Deliting it is safe. Following this until the algorithm terminates we have that $S = S^*$ and because the algorithm exhaustively searches the whole search tree this solution will be found.

2.

We have two cases. In the first case we include u and then delete it and all its neighbors from the graph. Since u has at least three neighbors, we delete at least four vertices. In the second case we only delete u. This gives us a branching of $n-4$ and $n-1$, or the branching vector $(4,1)$. Using the theorem from the lecture we have to solve $z^4 - z^3 - 1 = 0$ and obtain the largest (absolut value) solution, which is 1.3803. Therefore we get a runtime of $O(1.3803^n)$.