The class *SubExponential* (denoted SE) is defined as the class of search problems that have $2^{o(n)}$ time algorithms for input size n. Defining a *subexponential reduction family* or SERF, and the notion of SERF-reductions helps us understand which problems are in this class.[1]

1 SERF-Reductions

To build parameterized complexity, we need a stronger hypothesis than P $\neq$ NP.

Definition 8.1 (ETH - Exponential Time Hypothesis)**.** 3-SAT requires $2^{\Omega(n)}$ time.

The goal is to show that ETH $\implies$ k-Clique $\notin$ FPT.

Fact 8.2. *3*-SAT $\leq_p$ *3*-Coloring

Proof. Given an instance ϕ_1 of 3-SAT with n variables and m clauses, we obtain a 3-Coloring instance on a graph G with $O(m+n)$ vertices, constructed as shown in figure 8.1 (b).

- Construct $2n$ nodes for the variables and their negations, each pair connected by an edge.
- Construct a triangular palette with nodes F, T, N.
  They can be seen as assignments 0 (false), 1 (true) to the variables and a base vertex.
- Connect all variables and their negations to N.
- For a clause $x \vee y \vee z$, encode the OR gates as graph objects as shown in figure 8.1 (a).
- Connect the outputs of all OR objects to both F and N.

By construction, in a proper 3-coloring of G, u and $\neg u$ have different colors, and neither gets the same color as N. Same is true for F and T. Hence exactly one out of $u, \neg u$ gets the same color as F while the other gets the same color as T. As the outputs of all OR objects are connected to both N and F, all of them must get the same color as T in a proper 3-coloring. For a given proper coloring of G, assign 1 to exactly the variables with same color as T. This assignment satisfies ϕ_1. When ϕ_1 has a satisfying assignment, G has a proper 3-coloring by construction.

□

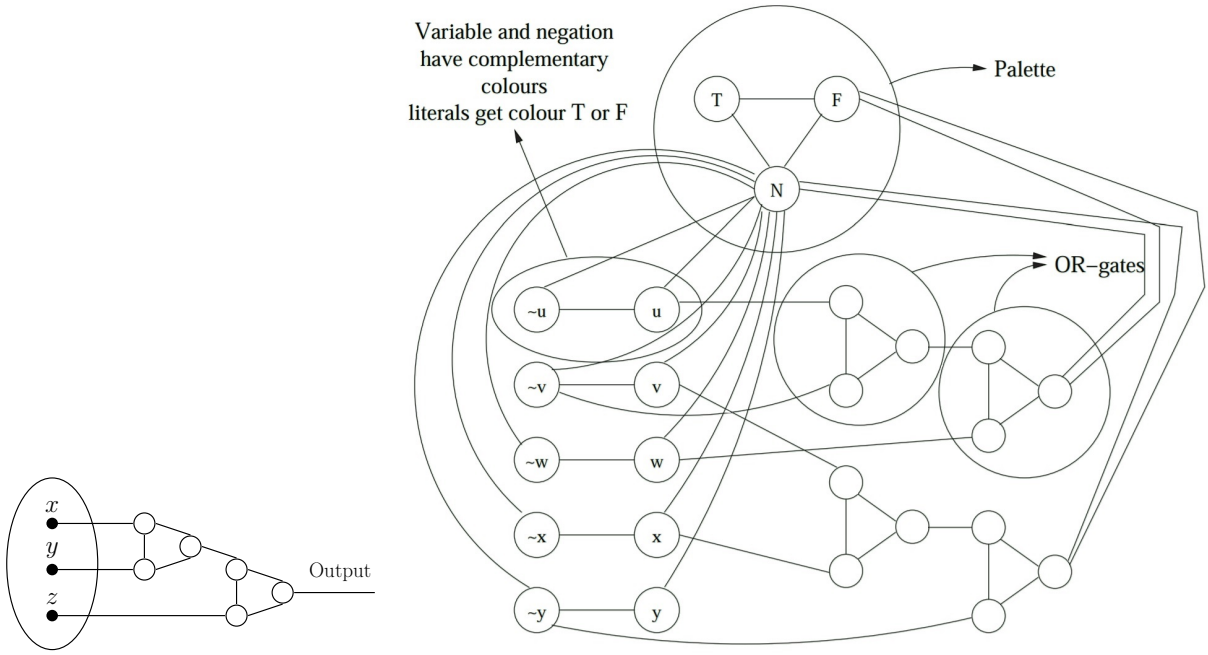
In the above construction, observe that m can be as large as n^3. Since G has $O(m+n)$ vertices,

$$\text{ETH} \implies \text{3-SAT} \notin \text{DTIME}\left(2^{o(n)}\right) \implies \text{3-Coloring} \notin \text{DTIME}\left(2^{o\left(n^{1/3}\right)}\right)$$

Fact 8.3. *3*-Coloring *requires* $2^{\Omega(n)}$ *time* $\implies$ k-Clique $\notin$ FPT.

Proof. Given a graph G, partition the vertex set into k parts, each with $\frac{n}{k}$ vertices as shown in figure 8.2. Consider proper 3-colorings of the induced subgraphs $G_1, G_2, .., G_k$ on these k parts with colors $\{1,2,3\}$ by going over the at most $3^{\frac{n}{k}}$ possibilities. Build a new graph G' with same vertex set as G as follows. For every edge $\{u,v\}$ in G, if u and v are colored differently and are not in the same part G_i, add the edge $\{u,v\}$ in G'. Note that G' has no edges inside any of the k partitions.

[1] Some parts based on lecture notes [Aro00]



(a) Graph object for clause $(x \vee y \vee z)$ (b) Graph G corresponding to 3-SAT formula, figure from [Pot20]

Figure 8.1: Reduction from 3-SAT to 3-Coloring

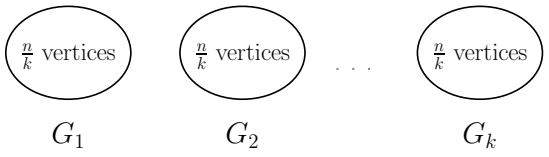


Figure 8.2: 3-Coloring to k-Clique reduction

As G_i's are induced subgraphs, each coloring of G_i's is compatible with G. If the constructed graph G' has a clique of size k, the vertices of this clique must lie in different G_i's and thus the colorings of $G_1, G_2, ..., G_k$ are pairwise compatible, giving a proper 3-coloring of G. This means if k-Clique is in FPT (time $f(k) \cdot n^c$ for a computable function f and constant c), then for any constant $k > 0$ we have a $O\left(3^{\frac{n}{k}} \cdot f(k) \cdot n^c\right) = O\left(3^{\frac{n}{k}} \cdot n^c\right)$ algorithm for 3-Coloring .

The hypothesis "3-Coloring requires $2^{\Omega(n)}$ time" means that there exists a universal constant $\epsilon > 0$ such that 3-Coloring requires $2^{\epsilon n}$ time. We can set $k := 2/\epsilon$, so that the algorithm runs in $O(3^{n/k} \cdot n^c) = O(3^{(\epsilon/2)n} n^c) \leq O(1.99^{\epsilon n})$ time, contradicting the hypothesis. □

[IPZ01] : Which problems have subexponential complexity?

Definition 8.4. Problem A is SERF-reducible (Sub-Exponential Reduction Family) to B (denoted $A \leq_{SERF} B$) if there exists a constant C such that for all $\epsilon > 0$, there is an oracle Turing machine $M = M_\epsilon$ such that

(i) M with oracle B (denoted M^B) solves A.

(ii) M runs in $O(2^{\epsilon ||x||_A})$ time on input x.

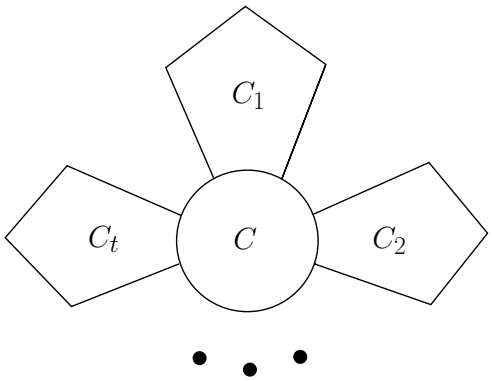


Figure 8.3: Sunflower obtained from $(C \vee C_1) \wedge (C \vee C_2) \ldots \wedge (C \vee C_t) \subseteq \phi$

(iii) Each query y that M makes to oracle B has length $||y||_B \leq C(||x||_A)$

where $||\ ||_A$ and $||\ ||_B$ are parameter values for problems A and B respectively.

Fact 8.5. *If $A \leq_{SERF} B$ and B can be solved in $O\left(2^{\delta||y||_B}\right)$ time for every $\delta > 0$ then A has a $O\left(2^{\delta'||x||_A}\right)$ algorithm for every $\delta' > 0$.*

Proof. Time taken for solving A is $O\Big(\underbrace{2^{\epsilon||x||_A}}_{\substack{\text{number of}\\ \text{oracle calls}}} \times \underbrace{2^{\delta C(||x||_A)}}_{\substack{\text{time for}\\ \text{solving B}}}\Big) = O\left(2^{(\epsilon+\delta C)||x||_A}\right)$.

□

2 The Sparsification Lemma

Theorem 8.6 (Sparsification Lemma [IPZ01])**.** *$\forall k \geq 3$, $\exists\ c_k \in \mathbb{N}$ such that*

$$k\text{-SAT} \ \leq_{SERF} k\text{-SAT}(c_k).$$

Proof. This proof uses a general form of backtracking, in which we pick clauses instead of variables and branch on them by satisfying some parts. This backtracking is also exponential time in the worst case, but we only use it for reduction from k-SAT to k-SAT(c_k).

Observe that when all variables of clause C_i are contained in clause C_j, then satisfying C_i satisfies C_j automatically. Thus clause C_j is redundant.

Definition 8.7. Given a k-SAT formula ϕ, $Simplify(\phi)$ removes all the redundant clauses in ϕ.

Definition 8.8 (Weak sunflower)**.** We call a set of clauses $C'_1, C'_2, ..., C'_t$ a weak sunflower, if $C = C'_1 \cap C'_2 ... \cap C'_t$ is nonempty. C is called the center and $\{C_i = C'_i \setminus C\}_{i=1}^t$ are called the petals of the sunflower. Each petal has the same size $(|C'_1| - |C|)$ and size of sunflower is t.

Note that the petals need not be disjoint in this case unlike the usual notion of sunflower, justifying the terminology 'weak sunflower'.

For a formula ϕ containing the sunflower in figure 8.3 to be satisfied, we need to either satisfy the center C, or each petal C_i. This gives the required branching for a clever backtracking algorithm.

We always pick a "large" sunflower and branch by adding the center or the petals as additional clauses, making the sunflower redundant. Each branching will look like figure 8.4, where C_j are

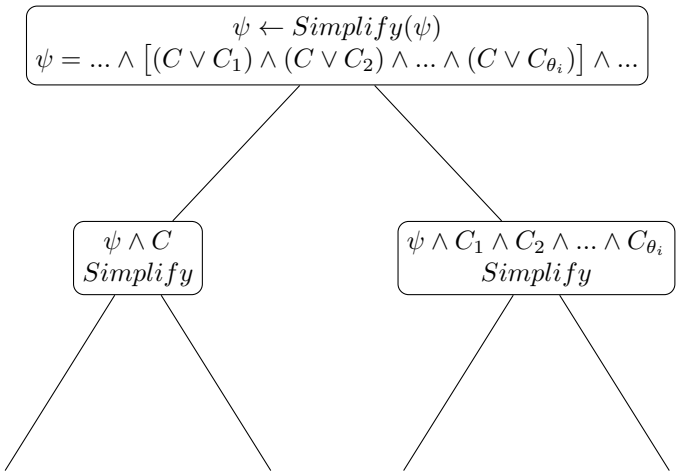


Figure 8.4: Center-Petal branching for a sunflower

petals of size i, C is the center of sunflower and θ_i is a large constant. If no sunflower can be picked, it means all variables occur in at most $\theta_i - 1$ clauses of size i, hence ϕ is an instance of k-SAT(c_k).

Algorithm 1 Reducing k-SAT to k-SAT (c_k)

procedure REDUCE(ϕ) ▷ ϕ is a k-SAT instance
  $\phi \leftarrow Simplify(\phi)$ ▷ Eliminate redundant clauses
  for $j = 2$ to k **do**
    for $i = 1$ to $j - 1$ **do**
      if there is a weak sunflower of size $t \geq \theta_i$
      with clauses $(C \vee C_1), (C \vee C_2), ..., (C \vee C_t)$ each of size j,
      each petal C_p of size i, and center C of size $j - i$ **then**
        REDUCE $(\phi \wedge C)$; ▷ Center addition
        REDUCE $(\phi \wedge C_1 \wedge C_2 \wedge ... \wedge C_t)$; ▷ Petal addition
        halt
      end if
    end for
  end for
  return ϕ ; **halt**
end procedure

Fact 8.9. *The algorithm maintains satisfiability of ϕ.*

Proof. If ψ_j are the leaves of REDUCE(ϕ), we clearly have $\phi \iff \bigvee \psi_j$. □

To prove that algorithm 1 gives a SERF-reduction from k-SAT, we show a bound of the form $2^{\epsilon n}$ on the number of leaves for any $\epsilon > 0$, and find constant c_k such that leaves are instances of k-SAT(c_k).

Fact 8.10. *Each leaf ψ is a c_k bounded occurrence, for $c_k = \sum_{i=1}^{k} (\theta_i - 1) \leq k\theta_k$.*

Proof. Since ψ is a leaf, no sunflower can be chosen in it. If a variable l was in at least θ_i clauses of size i, then the algorithm could branch on petals of size $i-1$ with center $\{l\}$. Thus every variable appears in at most $\theta_i - 1$ clauses of size i. Hence ψ is a c_k bounded occurrence for $c_k = \sum_i (\theta_i - 1)$. □

Fix a constant α such that $\frac{\alpha}{\log \alpha} > \frac{4k2^k}{\epsilon}$. Let $\beta_i = (4\alpha)^{2^{i-1}-1}$ and $\theta_i = \alpha\beta_i$. Notice that β_i are increasing with i and so are θ_i.

Procedure to bound the number of leaves.

1. Algorithm 1 adds $\leq \beta_{k-1} n$ clauses on any root-leaf path.
2. Number of petal-addition branches is very small ($\leq kn/\alpha$).
3. Number of leaves $\leq \sum_{r=0}^{kn/\alpha} \binom{\beta_{k-1} n}{r} \leq 2^{\epsilon n}$.

Lemma 8.11. *Suppose we add a clause of size i on a root-leaf path, then each variable appears in at most $2\theta_{i-1}$ clauses of size i in any branched child. (This also means that there can always be at most $2\theta_{i-1}$ many clauses of size i that were not originally in the k-SAT problem.)*

Proof. Consider the sunflower S that added clause(s) of size i at a step when the formula was ϕ'. S must consist of clauses of size $j > i$, since the i-clause(s) added is(are) either center or petal(s), each smaller than the clauses in S. ϕ' cannot have a sunflower of size θ_{i-1} consisting of i-clauses with petals of size $i-1$, since algorithm 1 would consider that first. Thus, each variable is in at most $\theta_{i-1} - 1$ clauses of size i in ϕ' before the center/petal(s) are added. If petals were added to ϕ', no variable v is in more than $\theta_{i-1} - 1$ petals. Because otherwise, algorithm would have considered center $C \cup \{v\}$ instead of C and added petals of size $i-1$. Thus when petals are added, no variable is in more than $2\theta_{i-1} - 2$ clauses of size i in branched child. If a center was added to ϕ', it's the only clause added, so lemma still holds.

□

Lemma 8.12. *An i-clause C' is said to be* eliminated *by a j-clause C ($j < i$) if during the $Simplify(\phi)$ step, C makes C' redundant. Consider any i-clause C' added by the algorithm at some step during its runtime (Thus C' is a center or petal added at some step). If C' is eliminated by a j-clause C ($j < i$), then C eliminates at most $2\theta_{i-1}$ clauses of size i.*

Proof. All clauses eliminated by C are eliminated in a single application of $Simplify$. By lemma 8.11, no variable is in more than $2\theta_{i-1}$ i-clauses, so set of variables in C can only be a subset of at most $2\theta_{i-1}$ of the i-clauses, thereby eliminating them.

□

Lemma 8.13. *The total number of clauses of size at most i added along any path is at most $\beta_i n$.*

Proof. Using lemma 8.12, we prove this by an induction on i. As there are at most n variables added along a path, $\beta_1 = 1$. Each added clause is either in the formula at a leaf, or gets eliminated by smaller clauses. The number of i-clauses in a leaf is at most $\theta_{i-1} n$. By the induction hypothesis, at most $\beta_{i-1} n$ clauses of size smaller than i are added along the path. By lemma 8.12, they eliminate at most $2\theta_{i-1}$ added i-clauses each. Thus total number of clauses of size at most i added along any path is at most

$$\beta_{i-1} n + \theta_{i-1} n + \beta_{i-1}(2\theta_{i-1}) n \leq 4\beta_{i-1}\theta_{i-1} n = \beta_i n.$$

The last equality comes from $\theta_{i-1} = \alpha\beta_{i-1}$, which gives

$$4\beta_{i-1}\theta_{i-1} = 4\alpha\beta_{i-1}^2 = 4\alpha\left((4\alpha)^{2^{i-2}-1}\right)^2 = 4\alpha\left((4\alpha)^{2^{i-1}-2}\right) = (4\alpha)^{2^{i-1}-1} = \beta_i.$$

□

Lemma 8.14. *There are at most kn/α petal additions along any path.*

Proof. Each addition of petals of size i adds at least θ_i petals. By lemma 8.13, at most $\beta_i n$ clauses of size i are added, so there are at most $\beta_i n/\theta_i$ petals additions with petal size i. Thus total number of petal additions is at most

$$\sum_i (\beta_i n/\theta_i) = \sum_i (n/\alpha) = kn/\alpha.$$

□

From lemma 8.13, we can possibly add at most $\beta_{k-1}n$ clauses, and on each node we add at least one petal of the sunflower or we add the center. Let $T(c,p)$ be the most leaves a tree can have if we're allowed to add c clauses and p petals. Then, we have the recursion for bound on the number of leaves as $T(c,p) = T(c-1,p-1) + T(c-1,p)$. This comes from the fact that when we do a petal addition, in the worst case, we might get to add only one petal. Otherwise a center is added, so number of petals allowed is the same. In either case, number of clauses allowed to be added decreases by at least one. The solution for this recurrence relation is $T(c,p) = \binom{c}{p}$ by the binomial identity $\binom{c-1}{p-1} + \binom{c-1}{p} = \binom{c}{p}$.

From lemma 8.14 we achieve the required bound on the total number of leaves as

$$\leq \sum_{r=0}^{kn/\alpha} \binom{\beta_{k-1}n}{r} \leq \left(\frac{\beta_{k-1}e\alpha}{k}\right)^{kn/\alpha} \leq \left((4\alpha)^{2^k}\right)^{kn/\alpha} = (4\alpha)^{\frac{k2^k n}{\alpha}} \leq 2^{\epsilon n}.$$

□

3 ETH implies k-CLIQUE is not in FPT

Corollary 8.15. 3-SAT $\leq_{SERF}$ 3-SAT(c_k) $\leq_{SERF}$ 3-Coloring .

The first inequality comes from theorem 8.6 and the second inequalities follows from the fact that the number of clauses in k-SAT(c_k) is linear in the number of variables. So a similar reduction as the one used in the proof of fact 8.2 results in a graph G with at most $O(n)$ vertices. Thus, we can SERF-reduce k-SAT(c_k) to 3-Coloring and that achieves our goal.

ETH	$\implies$	3-SAT requires $2^{\Omega(n)}$ time
	$\implies$	3-SAT(c_k) requires $2^{\Omega(n)}$ time
	$\implies$	3-Coloring requires $2^{\Omega(n)}$ time
	$\implies$	k-Clique $\notin$ FPT.

References

[Aro00] Sanjeev Arora. Exploring complexity through reductions. *IAS/Park City Mathematics Series*, https://www.cs.princeton.edu/~arora/pubs/aroraias.ps:Lecture 2 : 9–13, 00,0000. 1

[IPZ01] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001. 1, 8.6

[Pot20] Igor Potapov. 3-coloring is np-complete. *COMP309, Efficient Sequential Algorithms*, Slides https://cgi.csc.liv.ac.uk/~igor/COMP309/3CP.pdf(Class notes):Part 5, 2019-2020. 8.1b