Project 1 - Computationally Hard

Ari Gunnar Kristjónsson
Egill Magnússon

(b) determining if test01 is YES or NO

From the pattern `DDE`, we see that the substring `dde` must appear in $s = \texttt{abdde}$. This forces the assignments

$$D = \texttt{d}, \quad E = \texttt{e}.$$

Next, consider the patterns `ABD` and `ABd`. Since $D = \texttt{d}$, these require that `A B d` is a substring of s. The only match is `abd`, which forces

$$A = \texttt{a}, \quad B = \texttt{b}.$$

Finally, with these assignments the pattern `AAB` becomes `aab`. However, the string `aab` does not occur in $s = \texttt{abdde}$. Therefore, no consistent choice of expansions exists that satisfies all four patterns simultaneously.

$$\boxed{\text{Answer: NO}}$$

(c) description of the formal language used

The `.SWE` format itself can be seen as a formal language over the alphabet

$$\Sigma_{\text{SWE}} = \{0, 1, \ldots, 9\} \cup \{a, \ldots, z\} \cup \{A, \ldots, Z\} \cup \{:\} \cup \{,\} \cup \{\text{line break}\}.$$

A valid word in this language has the following structure:

1. The first line contains a natural number k, representing the number of patterns.
2. The second line contains the base string $s \in \{a, \ldots, z\}^*$.
3. The next k lines each contain a pattern $t_i \in \{a, \ldots, z, A, \ldots, Z\}^*$.
4. The remaining lines (at most 26) define the expansion sets. Each line is of the form

$$X : w_1, w_2, \ldots, w_\ell$$

   where X is an uppercase letter, and each $w_j \in \{a, \ldots, z\}^*$ is a possible expansion for X.

The *word problem* for this language is the following: given a file over the alphabet Σ_{SWE}, decide whether the file belongs to the `.SWE` format language. In practice, this means checking that the file obeys the rules above (correct first line, exactly k patterns, and correctly formatted expansion rules). If all conditions hold, the file is accepted as a valid input instance; otherwise, it is rejected.

(d) From decision to optimization

We assume we have access to an algorithm A_d for the decision version of SuperStringWithExpansion. That is, given an input instance, A_d returns `YES` if there exists a valid sequence of expansions, and `NO` otherwise.

We now construct an algorithm A_o for the optimization version, i.e. to actually output one valid expansion sequence $r_1, \dots, r_m$ or `NO` if none exists.

Algorithm A_o:

1. Run A_d on the original input. If the answer is `NO`, then return `NO`.
2. For each nonterminal γ_j with expansion set R_j:
   (a) For each candidate $r \in R_j$:
      i. Temporarily fix $\gamma_j := r$ and run A_d on the restricted instance.
      ii. If A_d returns `YES`, then permanently set $\gamma_j := r$ and continue with the next nonterminal.
   (b) If no candidate r works, return `NO`.
3. After all γ_j are fixed, output the chosen sequence $(r_1, \dots, r_m)$.

Correctness: - If A_d initially returns `NO`, then no solution exists, so A_o is correct. - Otherwise, a solution exists. When we test each candidate r for γ_j, keeping the one where A_d still answers `YES`, we guarantee that there is still a full solution consistent with the current choices. Proceeding through all nonterminals, we end with a full valid assignment.

Running time: - One initial call to A_d, plus at most $\sum_j |R_j|$ additional calls (each candidate tried once). - Since all expansion sets R_j are explicitly given in the input, the total number of calls is polynomial in the input size. - Each call to A_d counts as a single computational step by assumption, so A_o runs in polynomial time.

(e) SuperStringWithExpansion is in NP

Verifier.

- *Certificate:* A mapping that assigns each nonterminal (uppercase letter) $\gamma_j \in \Gamma$ one expansion string $r_j \in R_j$.
- *Verification procedure:*
  1. For each pattern t_i, replace each nonterminal with its chosen expansion to obtain the expanded string $e(t_i)$.
  2. Check whether $e(t_i)$ appears as a substring of the base string s.
  3. If all expanded patterns are substrings of s, return `YES`. Otherwise, return `NO`.
- *Correctness:*
  - If the instance is a YES-instance, then there exists some certificate mapping such that all patterns expand into substrings of s. For that mapping, the verifier accepts.
  - If the instance is a NO-instance, then for every certificate at least one expanded pattern is not a substring of s, so the verifier rejects.
- *Efficiency:* Replacing symbols and checking substrings can be done in polynomial time in the size of the input. Therefore, the verifier is polynomially bounded.

Since such a verifier exists, SuperStringWithExpansion is in NP.

(f) SuperStringWithExpansion is NP-complete

We reduce from Graph-3-Coloring. Given a graph $G = (V, E)$, we decide if V admits a proper 3-coloring with colors $\{r, g, b\}$.

Alphabets.

$$\Sigma = \{r, g, b, z\}, \quad \Gamma = \{\gamma_v \mid v \in V\}.$$

Here z is a separator symbol to prevent overlaps.

Expansion sets. For each $v \in V$:

$$R_v = \{r, g, b\}.$$

Master string. We define the base string as

$$s = \texttt{rgzrbzgrzgbzbrzbg}.$$

The substrings of the form xy in s are exactly the six ordered pairs with $x \neq y$. No rr, gg or bb occur. The separator z prevents unintended overlaps.

Patterns. For each edge $\{u, v\} \in E$, fix an arbitrary orientation and add

$$t_{\{u,v\}} := \gamma_u \gamma_v \in (\Sigma \cup \Gamma)^*.$$

and separate each $\gamma_u \gamma_v$ pair with a z.

Correctness. Suppose G is colorable. Let $c : V \to \{r, g, b\}$ be a proper coloring and choose $r_v := c(v) \in R_v$ for every v. For any edge $\{u, v\}$ with pattern $t_{u,v} = \gamma_u \gamma_v$, its expansion is $e(t_{\{u,v\}}) = c(u)c(v)$. Since $c(u) \neq c(v)$ this two letter pair is one fo the six xy substrings present in s. Hence every expanded pattern apears in s, so the SWE is a YES instance.

Suppose the SWE instance is YES, with choices $r_v \in \{r, g, b\}$. For any edge $\{u.v\}$, the expanded pattern is $r_u r_v$, which must be a substring of s. The only xy substrings in s have $x \neq y$, therefore $r_u \neq r_v$ for every edge. Defining $c(v) := r_v$ yields a proper 3-coloring of G. Thus G is 3-colorable.

Together, we have

$$G \text{ is 3-colorable} \iff \text{the constructed SWE instance is YES}$$

Size and time The instance size is $O(|V| + |E|)$: one γ_v and set $R_v = \{r, g, b\}$ per vertex one pattern per edge, and a constant length string s. Hence the reduction is in polynomial time.

Conclusion We have shown:

1. SWE $\in$ NP (in e))
2. Graph-3-coloring $\leq_p$ SWE via polynomial time, correctness preserviong reduction.

Therefore, SuperStringWithExpansion is NP-complete.

(g) Algorithm design

We design a simple brute-force algorithm that always gives the correct answer and always stops. It tries every possible combination of expansions for the uppercase letters and checks which one works. This approach has exponential worst-case running time but is guaranteed to find a solution if one exists.

Idea. Each uppercase letter $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_m\}$ has a set of possible expansions $R_1, R_2, \ldots, R_m$. The algorithm tests every possible choice of one word from each R_j. For each combination, we expand all patterns and check if every expanded pattern appears inside the base string s.

Algorithm.

1. **Input validation.** Read the input: base string s, all patterns $t_1, \ldots, t_k$, and all expansion sets $R_1, \ldots, R_m$. If any data is missing or malformed, immediately output `NO`.
2. **Try all combinations.** For every uppercase letter γ_j, choose one word r_j from R_j. Try all possible combinations of such choices.
3. **Expand patterns.** For each pattern t_i, replace every uppercase letter with its chosen expansion r_j to form the expanded string $e(t_i)$.
4. **Check substrings.** For each expanded $e(t_i)$, check whether it appears as a substring of s. If all expanded patterns are substrings of s, output the chosen expansions and stop.
5. **If no combination works, output `NO`.**

Heuristics. Although the algorithm always works, we can add some simple improvements:

- If a pattern with only lowercase letters is not a substring of s, output `NO` immediately.
- Remove any candidate $r \in R_j$ with $|r| > |s|$.
- Stop checking a combination as soon as one expanded pattern fails.

These checks can greatly reduce runtime on easy inputs but do not affect correctness.

Correctness. The algorithm is correct because:

- It only outputs `YES` when all expanded patterns actually appear in s.
- It explores every possible combination of expansions, so if any valid solution exists, it will be found.
- It halts after trying all finite combinations, so it always stops.

Running time. Let m be the number of uppercase letters and $|R_j|$ the number of expansions for each. The algorithm may try up to $\prod_{j=1}^{m} |R_j|$ combinations. This is exponential in the size of the input, but each check is polynomial, so the total runtime is bounded by $2^{p(n)}$ for some polynomial $p(n)$. This satisfies the requirement of an exponential but terminating algorithm.

(h) Worst-case running time analysis

Let m be the number of uppercase letters (nonterminals), $|R_j|$ the number of possible expansions for each γ_j, k the number of patterns, and $|s|$ the length of the base string.

In the worst case, the algorithm tests every possible combination of expansions:

$$N = \prod_{j=1}^{m} |R_j|.$$

For each combination, it expands all k patterns and checks whether each appears as a substring of s, which is polynomial in the input size.

Hence, the total running time is

$$T(n) = O\left(\left(\prod_{j=1}^{m} |R_j|\right) \cdot p(n)\right),$$

where $p(n)$ is a polynomial. The algorithm therefore has exponential worst-case running time but always halts and is correct for all inputs.

Division of labor

All parts of the project were done together. We collaborated closely on every task, discussing and implementing each part as a team rather than dividing the work.