1. In the Scheduling on Uniform Machines problem, known as $P||C_{\max}$ in the scheduling community, we are given *m machines*, *τ job types*, *processing times* $\{p_j\}_{j=1}^{\tau}$ which says how long does a job of type j run on any of the m machines, and *job counts* $\{n_j\}_{j=1}^{\tau}$ that says how many jobs with processing time p_j there are. A *schedule* of an input is the assignment of each job to a machine. The *duration* of a machine is the sum of processing time of jobs assigned to it. The *makespan* of a schedule is the maximum duration of the machines. The goal is to find a schedule of minimum makespan.

2. We will model $P||C_{\max}$ as an ILP. We guess the optimum makespan C and then write an integer program that finds whether there exists a schedule with makespan C or not. We will find C using binary search on the range $[0, \sum_{j=1}^{\tau} n_j p_j]$.

   Now let us write an ILP for $P||C_{\max}$ with makespan C. We shall have variables $x_{ij} \geq 0$ for $i \in [m]$ and $j \in [\tau]$ which says how many jobs of type j run on machine i. First we write a constraint that each machine runs finishes at time C, that is for every $i \in [m]$:

   $$\sum_{j=1}^{\tau} x_{ij} p_j \leq C \ .$$

   We also need a constraint that every job runs on some machine, that is for every $j \in [\tau]$ we have:

   $$\sum_{i=1}^{m} x_{ij} = n_j \ . \tag{1}$$

   We do not need an objective function, so we can set it to, e.g. max 0.

   Now we should observe that the ILP we wrote is actually an n-fold integer program as constraints (1) are independent between distinct j's. So that running time we get is $\mathsf{FPT}(\tau, \mathsf{m})$.

3. If s_i's and t_i's are not all different, then we can immediately answer with a no.

   First we observe that a solution exists if and only if there exists a solution that contains at most one non-endpoint vertex of every type. Since the paths are only needed to be vertex disjoint, then we can shortcut vertex repetitions within the same type. This is not true for edge-disjoint paths because there the shortcutting might cut through a different path.

   Let d be the neighborhood diversity of the input graph. The previous observation tells us that the vertex disjoint paths we are looking for can look in at most $2^d d^2$ ways depending on which vertices of the type graph it visits and which are the first and last vertices.

   We will write the following ILP: variables $x_{a,b,i}$ for $a, b \in [d]$ and $i \in [0, 2^d - 1]$ tells us how many paths there are in the solution that start in a vertex of type a, ends in a vertex of type b and only visits vertices of types that are 1 in the binary expansion of the number i. Let $c_{a,b}$ for $a, b \in [d]$ be the number of s_i, t_i pairs where s_i is of type a and t_i is of type b. For every $a, b \in [d]$ we need that the number of paths in the solution that start in a vertex of type a, ends in a vertex of type b is exactly equal to $c_{a,b}$, that is:

   $$c_{a,b} = \sum_{i=0}^{2^d - 1} x_{a,b,i} \ .$$

   If there is no way for a path to start in type a, end in type b and visit only vertices of type i, then also add a constraint $x_{a,b,i} \leq 0$ to force this selection to be unusable.

   Also the number of paths that use a vertex of type j must be at most the number of vertices of type j. Thus if n_j is the number of vertices of type j, then

   $$\sum_{a,b \in [d], i \in [0, 2^d - 1] \text{ such that the } j\text{-th bit of } i \text{ is } 1} x_{a,b,i} \leq n_j.$$

   Again, we do not care about the objective function. If this ILP is feasible, then there exists a solution which can be reconstructed greedily: for each path P_i with s_i of type a and t_i of type b, just look at an arbitrary $x_{a,b,i}$ that non-zero, greedily reconstruct the path so that it only uses vertices of whose type is 1 in the bit expansion of i, and decrement $x_{a,b,i}$.

Since there are $d^2 2^d$ variables, then we can use Lenstra's algorithm (running time $n^{\mathcal{O}(n)}$ for an n-variable integer linear program).

4. First observe that we can contract vertices of a type that forms an independent set into one as they can all share the same color.

   A *color category* is the set of colors which appears pre-colored in the same type. So for an example if we are doing 3-coloring, and we pre-color some vertices of a clique that defines a type with colors 1 and 2, then $\{1, 2\}$ is a color category. Categories are fixed by the input.

   A *color subcategory* further divides a color category: two colors in the same category are also in the same subcategory if they appear pre-colored in the same types. Note that a subcategory talks about colors, while categories only talks about colors used for pre-coloring.

   If a pre-coloring can be extended into a coloring, then we can divide all colors into subcategories.

   There are at most 2^{2d} subcategories. We can write an ILP with the following constraints.

   a) The number of colors of a category equals the sum of all subcategories in that category.

   b) The number of vertices of each type equals the sum of all subcategories which color a vertex of that type.

   Again we can solve it by Lenstra's algorithm.