This is a project for verifying the theorems and lemmas in topological network-control games played on graphs. All the codes are to compute the $Opt(G, C)$ in a brute-force way and then compare the result with the characterization provided by lemmas and theorems. If a lemma or theorem cannot cover some case, the counter case will be printed. Verifications are designed as follows:

Lemma 1:

- *lemma1_solver(opt_g, n)* is the function of computing the result by applying the lemma directly where *opt_g* is $Opt(G', \emptyset)$ and *n* corresponds to the same n in the lemma.
- In *lemma1_verifier*, we use dynamic programming algorithm to compute the result for all configurations $G' \cup Path_n$ with $Opt(G', \emptyset) \in [0, 1]$ and $n \leq 100$. Then after comparing the results with *lemma1_solver*'s results, the verification is done.

Lemma 2:

- *lemma2_solver(opt_g, n)* is the function of computing the result by applying the lemma directly where *opt_g* is $Opt(G', \emptyset)$ and *n* corresponds to the same n in the lemma.
- In *lemma2_verifier*, we use dynamic programming algorithm to compute the result for all configurations $G' \cup Path_n$ with $Opt(G', \emptyset) \in [2, 3]$ and $16 \leq n \leq 100$. Then after comparing the results with *lemma2_solver*'s results, the verification is done.

Corollary 3:

- *corollary3_solver(opt_g, n)* is the function of computing the result by applying the corollary directly where *opt_g* is $Opt(G', \emptyset)$ and *n* corresponds to the same n in the corollary.
- In *corollary3_verifier*, we use dynamic programming algorithm to compute the result for all configurations $G' \cup Cycle_n$ with $Opt(G', \emptyset) \in [2, 3]$ and $1 \leq n \leq 18$. Then after comparing the results with *corollary3_solver*'s results, the verification is done.

Theorem 1:

- *theorem1_solver(cycles)* is the function of computing the result by applying the theorem directly where *cycles[i]* is the size of $Cycle_{x_i}$, or namely, *cycles[i]* corresponds to the x_i in the theorem.
- *theorem1_generate* generates all the required cycles and *theorem1_compute* applies memorized searching algorithm to compute the results and stores them in *opt*.
- In *theorem1_verifier*, we apply *theorem1_generate* and *theorem1_compute* to generate and compute the result for configurations $(\bigcup Cycle_{x_i}, \emptyset)$ with $\sum x_i \leq 80$ and $x_i \in \bigcup_{j=0}^{9} P_j$. Then after comparing the results with *theorem1_solver*'s results, the verification is done.

Theorem 2:

- *theorem2_solver(cycles)* is the function of computing the result by applying the theorem directly where *cycles[i]* is the size of $Cycle_{x_i}$, or namely, *cycles[i]* corresponds to the x_i in the theorem.
- *theorem2_generate* generates all the required cycles and *theorem2_compute* applies memorized searching algorithm to compute the results and stores them in *opt*.
- In *theorem2_verifier*, we apply *theorem2_generate* and *theorem2_compute* to generate and compute the result for configurations $(\bigcup Cycle_{x_i}, \emptyset)$ of $\sum x_i \leq 60$. Then after comparing the results with *theorem2_solver*'s results, the verification is done.

Lemma 5:

- *lemma5_solver(opt_g, n, m)* is the function of computing the result by applying the lemma directly where *opt_g* is $Opt(G', \emptyset)$ and *n*, *m* correspond to the same n, m in the lemma.
- In *lemma5_verifier*, we use dynamic programming algorithm to compute the result for all configurations $G' \cup Path_n \cup Path_m$ with $Opt(G', \emptyset) \in [0, 1]$ and $1 \leq n, m \leq 10$. Then after comparing the results with *lemma5_solver*'s results, the verification is done.

Lemma 7:

- *lemma7_solver(opt_g, n, m)* is the function of computing the result by applying the lemma directly where *opt_g* is $Opt(G', \emptyset)$ and *n*, *m* correspond to the same n, m in the lemma.
- In *lemma7_verifier*, we use dynamic programming algorithm to compute the result for all configurations $G' \cup Path_n \cup Path_m$ with $Opt(G', \emptyset) \in [2, 3]$ and $16 \leq n \leq 25, m \leq 25$. Then after comparing the results with *lemma7_solver*'s results, the verification is done.

Lemma 8:

- *lemma8_solver(opt_g, n)* is the function of computing the result by applying the lemma directly where *opt_g* is $Opt(G', \emptyset)$ and *n* correspond to the same n in the lemma.
- In *lemma8_verifier*, we use dynamic programming algorithm to compute the result for all configurations $G' \cup Path_n$ with $Opt(G', \emptyset) \in [2, 3]$ and $1 \leq n \leq 33$. Then after comparing the results with *lemma8_solver*'s results, the verification is done.

Theorem 3:

- *theorem3_solver(paths)* is the function of computing the result by applying the theorem directly where *paths[i]* is the size of $Path_{x_i}$, or namely, *paths[i]* corresponds to the x_i in the theorem.
- *theorem3_generate* generates all the required paths and *theorem3_compute* applies memorized searching algorithm to compute the results and stores them in *opt*.
- In *theorem3_verifier*, we apply *theorem3_generate* and *theorem3_compute* to generate and compute the result for configurations $(\bigcup Path_{x_i}, \emptyset)$ with $\sum x_i \leq 75$ and $x_i \in \bigcup_{i=0}^{7} P_i$. Then after comparing the results with *theorem3_solver*'s results, the verification is done.

Theorem 4:

- *theorem4_solver(paths)* is the function of computing the result by applying the theorem directly where *paths[i]* is the size of $Path_{x_i}$, or namely, *paths[i]* corresponds to the x_i in the theorem.
- *theorem4_generate* generates all the required paths and *theorem4_compute* applies memorized searching algorithm to compute the results and stores them in *opt*.
- In *theorem4_verifier*, we apply *theorem4_generate* and *theorem4_compute* to generate and compute the result for configurations $(\bigcup Path_{x_i}, \emptyset)$ of $\sum x_i \leq 60$. Then after comparing the results with *theorem4_solver*'s results, the verification is done.

Other functions:

- *path_p_type(path)* computes which P_i the path belongs to.
- *get_path_cntp(paths)* computes $cnt_{P_i}(G)$ where G is a linear forest defined by *paths*.
- *get_path_cntodd(paths)* computes $cnt_{odd}(G)$ where G is a linear forest defined by *paths*.
- *get_path_s(paths)* computes the $\mathcal{S}(G)$ where G is a linear forest defined by *paths*.
- *cycle_p_type(cycle)* computes which P_i the cycle belongs to.
- *get_cycle_cntp(cycles)* computes $cnt_{P_i}(G)$ where G is union of cycles defined by *cycles*.
- *get_cycle_cntodd(cycles)* computes $cnt_{odd}(G)$ where G is union of cycles defined by *cycles*.

- *get_cycle_s(cycles)* computes the $\mathcal{S}(G)$ where G is union of cycles defined by *cycles*.

How to run the code?

- Firstly, install the rust environment with command `curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh` (see https://www.rust-lang.org/tools/install)
- Then run with command `cargo run --release` in the work directory containing *Cargo.toml* file.
- In some time, you will see the following result printed on the terminal:

```
Verifying Lemma 1 for 1<=n<=100 ..
Lemma 1 passed!
Verifying Lemma 2 for 16<=n<=100 ..
Lemma 2 passed!
Verifying Corollary 3 for 1<=n<=18 ..
Corollary 3 passed!
Verifying Theorem 1 for 1<=|G|<=80 ..
Theorem 1 for |G|<=80 passed!
Verifying Theorem 2 for 1<=|G|<=60 ..
Theorem 2 for |G|<=60 passed!
Verifying Lemma 5 for 1<=n,m<=10 ..
Lemma 5 passed!
Verifying Lemma 7 for 16<=n<=25 and m<=25 ..
Lemma 7 passed!
Verifying Lemma 8 for 1<=n<=33 ..
Lemma 8 passed!
Verifying Theorem 3 for |G|<=75 ..
Theorem 3 for |G|<=75 passed!
Verifying Theorem 4 for |G|<=60 ..
Theorem 4 for |G|<=60 passed!
```

The M2 Max completed the task in 10 minutes, whereas the i7-12700H took 50 minutes, highlighting distinct performance variations among the different CPUs.