**Как «Forward Proposer» генерирует синтетические теоремы**

*(пересказ статьи “Training a First‑Order Theorem Prover from Synthetic Data” на русском)*

1. **Отправная точка — аксиомы предметной области**  
   Для каждой из десяти выбранных областей TPTP (теория полей, геометрия, теория групп и т. д.) берут набор аксиом в клаузальной форме (CNF).
2. **Линейная резолюция: строим «цепочку» выводов**
   * Пусть C₀ … C\_N — последовательность клауз.
   * C₀ выбирают из аксиом.
   * На каждом шаге *t* = 1…N текущую клаузу C\_{t‑1} **обязательно** используют в следующей резолюции или факторизации, получая C\_t.
   * Так как новая клауза всегда участвует в следующем выводе, дерево вывода вырождается в линию (linear resolution), но при этом остаётся полнотой по Ли.
3. **Стохастический выбор вывода с «температурой»**  
   Простая равномерная выборка быстро порождает гигантские клаузулы. Поэтому каждый возможный вывод оценивают по размеру результирующей клаузулы (числу символов) и выбирают через soft‑max

P(i)=exp⁡(−∣Ci∣/T)∑jexp⁡(−∣Cj∣/T), P(i)=\frac{\exp(-|C\_i|/T)}{\sum\_j\exp(-|C\_j|/T)},P(i)=∑j​exp(−∣Cj​∣/T)exp(−∣Ci​∣/T)​,

где **T** — температура: низкая T усиливает предпочтение к компактным формулам, высокая T даёт больше разнообразия.

1. **Через N шагов получаем гипотезу**  
   После ровно **N** резолюций итоговая клауза C\_N объявляется *конъюнктурой* задачи. Поскольку она выведена из аксиом, импликация

Аксиомы  ⊢  CN \text{Аксиомы} \;\vdash\; C\_NАксиомы⊢CN​

гарантированно истинна.

1. **Подбор параметров N и T по трём критериям**  
   Для каждой области проводят сеточный поиск (N, T) и генерируют 1 млн теорем; затем выбирают сочетание, которое:
   * максимально затрудняет доказательство для E‑prover (сложность),
   * удерживает средний размер клаузы ≤ 64 символов (компактность),
   * обеспечивает ≥ 500 000 уникальных теорем из миллиона (разнообразие).

Пример: для GEO8 взяли N = 10, T = 8.

1. **Поток данных для обучения**  
   С выбранными (N, T) «Forward Proposer» генерирует десятки миллионов теорем, которые служат опытом для нейронного резолюционного доказателя.

**Почему подход работает**

* **Звукoвость по определению** — каждая синтетическая теорема уже доказана построением.
* **Управляемый уровень сложности** — N регулирует глубину вывода, T — сложность формул.
* **Широкий охват домена** — используются только символы аксиом, но пространство теорем гораздо шире, чем в TPTP.

**Ограничения, отмеченные авторами**

* Генерируются только теоремы, сводимые к **одной клаузе** (без ∃‑кванторов и новых символов); это покрывает ≈ 78 % задач TPTP.
* Могут появляться «скучные» или малоинформативные утверждения; нужна мера *интересности*.
* Линейная резолюция не порождает ветвящихся деревьев доказательств.

Если хотите поэкспериментировать:

* реализуйте простой цикл на 10 шагов с soft‑max выбором вывода;
* измеряйте сложность своим доказателем и подбирайте N/T;
* чтобы покрыть ∃‑теоремы, разрешите выводить не одну, а набор клауз или добавляйте свежие сколем‑константы.

Готов предоставить пример кода, идеи расширения на равенство или советы по интеграции с вашим исследованием разрешения и ML — дайте знать!

create\_examples.py

def all\_resolvents(C1, C2):

Ниже приведён один из возможных вариантов реализации функции forward\_propose, которая создаёт линейную цепочку клауз из исходных аксиом. Эта функция выбирает случайную стартовую клаузу (C₀), а затем на каждом шаге пытается разрешить последнюю полученную клаузу с каждой из уже известных, используя метод all\_resolvents (мы пока его оставляем как внешнюю зависимость – его вы реализуете позже). Результат выбирается через softmax‑взвешивание, где более короткие клаузы получают больший вес.

**Пояснения:**

1. **Инициализация**  
   Функция берёт все аксиомы (role 'axiom') из списка, полученного от parse\_tptp\_clauses, и выбирает случайную из них для старта цепочки.
2. **Цикл по шагам (t от 0 до N-1)**  
   На каждом шаге выбираем резолвенты для последней клаузы (chain[-1]) с каждой из известных клауз. Все найденные кандидаты собираются в список.
3. **Выбор кандидата**  
   Если список кандидатов не пуст, то по softmax‑взвешиванию (с весами, обратными размеру клаузы) выбирается одна резолвента, которая добавляется как новая клауза в цепочку и в список известных клауз.
4. **Возврат результата**  
   После завершения всех шагов функция возвращает последнюю клаузу, которая станет заключительной теоремой (conjecture) для дальнейшей генерации задачи.

Resolvable Pairs Count

| **What you store** | **What the network will learn** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| **Only the best pair** (one positive example per problem) | A pure *generation* / *pointer* task: given the clause set, output one pair of literals. | • Compact dataset. • No risk of overwhelming the learner with negatives. | • No direct negative signal – the model never sees what a *bad* pair looks like. • You’ll need to generate candidates at training time anyway (to calculate a loss such as margin‑ranking or cross‑entropy over the candidate set), which is slower and pushes work into the training loop. |
| **Best pair + *all* other candidate pairs (labelled 1 / 0)** | A *ranking / classification* task: score each candidate pair so the best ranks highest. | • Clear positive vs. negative supervision. • Candidate enumeration happens once when you build the dataset, so training is fast. | • For some problems the number of negatives can explode (thousands of pairs). That inflates disk‑space, memory, and may harm class balance. |
| **Best pair + *sub‑sampled* negatives** | Same as above, but you keep only *k* negatives per problem (e.g. 20 random or heuristic‑hard negatives). | • Keeps explicit negative signal. • Dataset size stays reasonable. • You can bias the negatives toward “hard” ones (same predicate, similar arguments, etc.). | • Requires a sampling step in preprocessing. |

 **You almost always need negatives**  
A model that never sees wrong answers will struggle to distinguish right from wrong. Even if you adopt a pointer‑style architecture (no explicit candidate list), you will still compute a loss that contrasts the chosen pair with *something* negative (e.g. all other literals or a sampled subset). So it helps to have negatives pre‑computed.

 **Keep them, but trim**

* Store the **clauses** in every example (they’re essential).
* Keep **one positive pair** and, say, 10‑50 negatives per problem. You can:
  + Choose negatives uniformly at random, or
  + Prefer “hard” negatives (same predicate, opposite sign but arguments almost unifiable, etc.). Hard negatives give stronger gradients.

extract\_solutoin

Here’s a self‑contained Python snippet that will:

1. **Scan every** \*\_rs.jsonl in your Res\_Pairs folder
2. **Extract the first original inference** from the matching \*\_solved.txt (using “axiom”/“negated\_conjecture” only)
3. **Map each proof‑clause** back to its index in the JSONL “clauses” array by matching its clause‑body text
4. **Look for an existing resolvable‑pair** entry with exactly those two clause‑indices
5. **If absent**, regenerate the candidate pairs for that clause‑pair and append the first one
6. **Finally**, rewrite the JSONL so the last line becomes