The-sudoku-9x9-solver — ветка 2v

Кратко
Проект создан для личного обучения и пополнения резюме (цель — позиция backend-разработчика). Система находит поле Sudoku на экране, распознаёт цифры в каждой клетке, запускает C++ решатель и автоматически заполняет поле. Эта ветка (`2v`) объединяет детектор поля (YOLO), классификатор цифр и классический C++ решатель.

Цели ветки

- Найти на экране область с 9×9 полем Sudoku (детектор YOLO).
- Вырезать 81 ячейку и распознать, какая цифра в каждой клетке (классификатор на базе PyTorch / ResNet, веса `best.pt`).
- Передать текущее заполнение в C++ решатель (`a.exe` / скомпилированный `sudoku.cpp`), получить решение и автоматически ввести правильные цифры в интерфейс игры.

Дерево файлов (реальные файлы в ветке 2v)

```
/ (корень ветки 2v)
├─ .gitattributes
├─ README.md                # (этот файл)
├─ a.exe                    # предкомпилированный бинарник-решатель (Windows)
├─ best.pt                  # веса классификатора цифр (digit classifier)
├─ best.pth                 # веса YOLO (detector для поиска Sudoku на экране)
├─ recogniser.py            # основной Python-скрипт запуска пайплайна
├─ sudoku.cpp               # исходник C++ решателя
```

Что делает `recogniser.py` — пошагово (ключевые моменты)

1. **Захват экрана и поиск поля**
    - Делает глобальный скриншот экрана (PIL `ImageGrab`) и подаёт в YOLO модель, загруженную из `best.pth` (через `ultralytics.YOLO`).
    - Получает bounding box поля: `x1,y1,x2,y2`. Если детектор не дал бокса — можно включить ручной режим (ввести координаты мышью).
2. **Вырезание 9×9 клеток**
    - Делит bbox на равные 9×9 ячеек, кропит каждую ячейку как изображение.
3. **Распознавание цифр (классификатор)**

- Каждую ячейку подаёт в классификатор (веса `best.pt`). Классификатор — ResNet-подобная модель, загруженная из чекпойнта. Возвращает метку (1..9) или `empty`/marker для пустой клетки.
- Результат преобразуется в последовательность 81 токена (например: `5 3 # # 7 ...`) — формат пустого маркера и разделители согласованы между Python и C++.

4. **Вызов C++ решателя**

   - Python формирует командную строку и вызывает бинарник: `a.exe token1 token2 ... token81` (или `./a.out` на Linux) с помощью `subprocess.run`/`os.system`.
   - Решатель парсит вход, решает судоку и записывает решение в `out.csv` (или выводит в stdout — предпочтительнее stdout).

5. **Чтение решения и автозаполнение**

   - Python ждёт завершения решателя, читает `out.csv` (или stdout), получает решённую матрицу.
   - Эмулирует ввод в UI игры (mouse/keyboard): перемещает курсор по ячейкам и вводит цифры.

Детальный разбор `sudoku.cpp` (алгоритм, структуры, оптимизации)

Ниже — технический разбор алгоритма, реализованного в `sudoku.cpp`. Это подробное руководство по логике, структурам и возможным улучшениям.

1) Идея алгоритма (высокоуровнево)

- **Propagation (итеративное упрощение):** для каждой пустой клетки вычисляются кандидаты (числа, не встречающиеся в её строке/столбце/блоке). Если у клетки ровно один кандидат — фиксируем его и повторяем (это может уменьшить кандидатные множества соседей). Повторяем, пока добавления есть.
- **Backtracking (предположения):** когда более нет однозначных клеток, выбирают клетку с наименьшим числом кандидатов (MRV — Minimum Remaining Values) и делают предположение — ставят одну из кандидатур и рекурсивно продолжают. При противоречии (ноль кандидатов) откатываемся и пробуем следующий кандидат. Стек предположений хранит информацию для отката.

2) Вход/выход

- **Вход:** 81 токен (1..9 или маркер пустой), обычно передаётся аргументами командной строки.
- **Выход:** решённая 9×9 матрица — в `out.csv` или в stdout.

3) Основные структуры данных

- `grid[9][9]` — текущая сетка (0 — пустая).
- `candidates[9][9]` — множество возможных цифр для каждой клетки.

- `stack` (предположений) — элементы: (r, c, tried_values, snapshot_id) или журнал изменений для отката.

4) Псевдокод решения

```
function solve(grid):
    compute candidates for all empty cells
    while any cell has exactly one candidate:
        place that candidate and update neighbors' candidates
        if any neighbor now has 0 candidates: return False
    if grid complete: return True
    pick (r,c) with minimal |candidates| > 1
    for v in candidates[r][c]:
        snapshot = save_state()
        place v; update candidates
        if solve(grid): return True
        restore_state(snapshot)
    return False
```

Как запускать (быстрая инструкция, Windows)

1. Клонировать ветку `2v`:

```
git clone --branch 2v https://github.com/aRTIKa-afk/The-sudoku-9x9-solver-TS9S.git
cd The-sudoku-9x9-solver-TS9S
```

2. Создать виртуальное окружение и установить зависимости:

```
python -m venv .venv
.venv\Scripts\activate   # Windows
pip install -r requirements.txt
```

3. (Опционально) Скомпилировать `sudoku.cpp` локально (Windows / Linux):

- Windows (MSVC): `cl /O2 /std:c++17 sudoku.cpp /Fe:a.exe`
- Linux: `g++ -O2 -std=c++17 sudoku.cpp -o a.out` (и в `recogniser.py` заменить вызов `a.exe` на `./a.out`)

4. Запустить основной пайплайн:

```
python recogniser.py
```