następnie zostanie pomieszana. „Losowe” układanki należy generować, rozpoczynając od ułożonej planszy i wykonując zadaną w konstruktorze *liczbę mieszań*, nie dbając o ewentualne niwelowanie się ruchów przeciwnych, natomiast dbając o nie zliczanie się ruchów pustych przy brzegach planszy.

E **Ćwiczenie 2.16** **Porównaj działanie algorytmów A* i Best-first search rozwiązujących „puzzle przesuwne”.** Wskazówki: wykonaj eksperyment statystyczny (analogicznie jak w Ćwiczeniu 2.15) rozwiązując każdą z plansz początkowych dwukrotnie algorytmami A* i Best-first search (przy ustalonej heurystyce). W ramach eksperymentu obserwuj: średni czas, średnią liczbę odwiedzanych stanów i średnią długość znalezionej ścieżki.

E **Ćwiczenie 2.17** **Zaimplementuj trzecią heurystykę „Manhattan + konflikty liniowe” do programu rozwiązującego „puzzle przesuwne”.** Wskazówki: zaimplementuj dodatkową trzecią funkcję heurystyczną „Manhattan + konflikty liniowe”, która doliczy do podstawowego składnika Manhattan dwa ruchy za każdy obecny na planszy konflikt liniowy (uwaga: zgodnie z informacjami podanymi w sekcji 2.3.2, zlicz konflikty liniowe w wierszach i kolumnach bez nadmiarowości). Porównaj działanie nowej heurystyki z poprzednimi poprzez odpowiedni eksperyment statystyczny (analogicznie jak w Ćwiczeniu 2.7).

## 2.6 Ćwiczenia laboratoryjne (C++ + biblioteka *SI++*)

E **Ćwiczenie 2.18** **Napisz program rozwiązujący łamigłówkę sudoku z wykorzystaniem algorytmu Best-first search i heurystyki „liczba niewiadomych”.** Wskazówki: napisz klasę szablonową `generic_sudoku` reprezentującą stan planszy sudoku o rozmiarze $MN \times MN$ ($M$ to liczba wierszy małego podprostokąta, a $N$ to liczba kolumn małego podprostokąta; wymiary przekaż w parametrach szablonu) i zadanej heurystyce — wybierz odpowiedni typ tablicowy (tablica tablic `std::array`); wykonaj dziedziczenie z klasy `graph_state`; przygotuj konstruktor przyjmujący tablicę reprezentującą planszę; dostarcz implementacje metod:

- `clone()` — zwraca wskaźnik na obiekt będący kopią obiektu `this`,
- `hash_code()` — na podstawie wypełnienia planszy generuje liczbę stanowiącą indeks do tablicy mieszającej; można oprzeć się na kodzie źródłowym metody `hashCode()` klasy `String` w języku Java,
- `get_successors()` — generując stany potomne wybierz dowolną pustą komórkę; wolno generować wyłącznie poprawne stany,
- `is_solution()` — jeśli znajduje się pusta komórka, to potraktuj planszę jako nierozwiązaną,
- `to_string()`,
- `is_equal()` (w sekcji `protected`),

⋆ `get_heuristic_grade()` (w sekcji `protected`) — wartość zadanej heurystyki;

napisz właściwy program w funkcji `main()`, a w nim stwórz początkowy stan sudoku zasilony z tablicy (rozważ napisanie konstruktora, który przyjmie napis reprezentujący diagram sudoku) i uruchom rozwiązywanie konstruując obiekt klasy `informative_searcher`, któremu w konstruktorze oprócz stanu początkowego przekażesz komparator; sprawdź poprawność działania dla kilku przykładów; poza rozwiązaniem wypisz dodatkowo na ekran informacje na temat: czasu rozwiązywania, liczby stanów w zbiorach *Open* i *Closed* w chwili stopu; same klasy mogą wyglądać następująco:

```
template<int M, int N, typename Heuristic>
class generic_sudoku : public graph_state
{
public:
    // implementacje metod...

protected:
    double get_heuristic_grade() const override
    {
        return heuristic(board);
    }

    std::array<...> board;
    static constexpr Heuristic heuristic {};
};

template<int M, int N>
struct H_remaining
{
    double operator()(const std::array<...> &board) const
    {   // TODO
        return 0;
    }
};
```

komparator można zdefiniować następująco:

```
auto comp = [](const graph_state &a, const graph_state &b)
{
    return a.get_h() < b.get_h();
};
```

metoda `is_equal()` może wyglądać tak:

```
bool is_equal(const graph_state &s) const override
{
    const generic_sudoku *st = dynamic_cast<const
        generic_sudoku*>(&s);
    return st != nullptr && st->board == this->board;
}
```

zaś metoda `get_successors` może mieć następujący zarys:

```
std::vector<std::unique_ptr<graph_state>> get_successors()
    const override
{
    std::vector<std::unique_ptr<graph_state>> successors;
    for (int i = 0; i < M*N; i++)
        for (int j = 0; j < M*N; j++)
            if (board[i][j] == 0)
            {
                for (int8_t p : possibilities(i, j))
                {
                    auto c = clone();
                    ((generic_sudoku&)*c).board[i][j] = p;
                    c->set_parent(this);
                    c->update_score(get_g() + 1);
                    successors.push_back(std::move(c));
                }
                return successors;
            }
    return {};
}
```

gdzie metoda `possibilities` zwraca dopuszczalne liczby, które można wstawić do kratki, tak by nie zaburzały ograniczeń. Współrzędne lewego górnego rogu podprostokąta, do którego należy pole o współrzędnych $(i, j)$, wynoszą $\left(\left\lfloor\frac{i}{M}\right\rfloor \cdot M, \left\lfloor\frac{j}{N}\right\rfloor \cdot N\right)$ — skorzystaj z nich, by przeiterować po elementach podprostokąta w celu wykluczenia liczb.



**Ćwiczenie 2.19** **Modyfikując odpowiednio początkową planszę sudoku znajdź więcej niż jedno rozwiązanie (wykorzystaj program z Ćwiczenia 2.18).** Wskazówka: konstruktor klasy `informative_searcher` przyjmuje trzeci parametr określający liczbę rozwiązań do znalezienia — przekaż wartość `std::numeric_limits<size_t>::max()`.

**Ćwiczenie 2.20** **Wyznacz liczbę wszystkich rozwiązań sudoku dla planszy $6 \times 6$ — $M = 2, N = 3$ (wykorzystaj program z Ćwiczenia 2.18).** Wskazówka: dokonaj obliczeń w sposób pośredni, tzn. wyznacz liczbę rozwiązań dla planszy, której pierwszy wiersz zawiera cyfry `1,2,3,4,5,6` (wykorzystaj doświadczenia z Ćwiczenia 2.19), a uzyskaną liczbę rozwiązań przemnóż przez wartość `6!` (liczba permutacji cyfr wiersza).

**Ćwiczenie 2.21** **Ulepsz program rozwiązujący sudoku poprzez generowanie potomków w „komórce minimalnej”.** Wskazówki: stwórz nową klasę dziedziczącą po

`generic_sudoku` i podmień implementacje metod `clone()` i `get_successors()` (w niej znajdź komórkę z najmniejszą liczbą możliwości wypełnienia i wygeneruj jej potomków; porównaj działanie nowej i starej wersji programu rozwiązującego (czasy wykonania, liczba odwiedzanych stanów). Do odziedziczonego składnika `board` w klasie pochodnej trzeba odwołać się w sposób jawny za pomocą wskaźnika `this` (bądź za pomocą operatora zakresu), ponieważ klasa `generic_sudoku` jest klasą szablonową.

E **Ćwiczenie 2.22 Zaimplementuj dodatkową heurystykę „suma pozostałych możliwości" do programu rozwiązującego sudoku.** Wskazówki: stwórz klasę podobną do `H_remaining`; porównaj działanie obu heurystyk dla kilku przykładów (czasy wykonania, liczba odwiedzanych stanów); przygotuj większy eksperyment statystyczny w ramach funkcji `main()`, który porówna dwie heurystyki dla przynajmniej 100 przykładów.

E **Ćwiczenie 2.23 Napisz program rozwiązujący układankę „puzzle przesuwne" z wykorzystaniem algorytmu A* oraz heurystyk „kafelki na niewłaściwym miejscu" i „Manhattan".** Wskazówki: stwórz generyczną klasę `sliding_puzzle` reprezentującą stan planszy układanki „puzzle przesuwne", postępując zgodnie z ogólnymi wytycznymi z Ćwiczenia 2.18 (parametryzowana wymiarowość, konstruktory, dziedziczenie z klasy `graph_state`, itd.); przygotuj metodę generującą pomieszaną planszę (do wielokrotnego wykonywania losowych ruchów użyj obiektów klasy `std::default_random_engine` oraz `std::uniform_int_distribution`); przygotuj klasy reprezentujące heurystyki „kafelki na niewłaściwym miejscu" i „Manhattan"; przygotuj dwie wersje funkcji `main()` — wariant pierwszy pozwalający rozwiązać pojedynczą układankę za pomocą algorytmu A* (obiekt klasy `informative_searcher` z odpowiednim komparatorem) i wypisać dla niej ścieżkę ruchów (przygotuj statyczną metodę, która przyjmie jako parametr wskaźnik na rozwiązanie, a w wyniku zwróci napis przedstawiający ruchy), oraz wariant drugi wykonujący statystyczne porównanie dwóch heurystyk; w ramach drugiego wariantu wygeneruj 100 losowych plansz początkowych (każda pomieszana za pomocą 1000 ruchów) i każdą z nich rozwiąż dwukrotnie, oblicz i wyświetl średnią liczbę stanów odwiedzanych przez każdą z heurystyk oraz średnie czasy wykonań; komparator można zdefiniować następująco:

```
auto comp = [](const graph_state &a, const graph_state &b)
{
    return a.get_f() < b.get_f();
};
```

E **Ćwiczenie 2.24 Porównaj działanie algorytmów A* i Best-first search rozwiązujących „puzzle przesuwne".** Wskazówki: wykonaj eksperyment statystyczny (analogicznie jak w Ćwiczeniu 2.23) rozwiązując każdą z plansz początkowych dwukrotnie algorytmami A* i Best-first search (przy ustalonej heurystyce); w ramach porównania obserwuj: średni czas, średnią liczbę odwiedzanych stanów i średnią długość znalezionej ścieżki.

E **Ćwiczenie 2.25 Zbadaj wpływ zmiany porządku odwiedzania stanów o równej wartości $f$.** Wskazówka: przygotuj komparator porównujący dwa stany `a` i `b` i zwracający prawdę, gdy $f_a < f_b \vee f_a = f_b \wedge h_a < h_b$; porównaj działanie nowego sposobu porządkowania z poprzednim poprzez odpowiedni eksperyment statystyczny (analogicznie jak w Ćwiczeniu 2.23).