Laboratorium 2
Wprowadzenie do sztucznej inteligencji

Agnieszka Głuszkiewicz

1 Opis zadania

Celem zadania było zaimplementowanie **algorytmu A*** do rozwiązywania układanki logicznej *Piętnastka*. W mojej implementacji zastosowałam algorytm **Bidirectional A***. Równocześnie przeszukuję od stanu początkowego do docelowego oraz od stanu docelowego do początkowego, spotykając się w środku, co często znacząco redukuje przestrzeń przeszukiwania. Skupiłam się na dwóch heurystykach: **Manhattan** oraz **Linear Conflict**.

2 Opis heurystyk

2.1 Manhattan

Heurystyka odległości Manhattan to suma odległości w pionie i poziomie, jaką każdy kafelek musiałby przebyć z obecnej pozycji do swojej pozycji docelowej.

$$h_{\text{Manhattan}}(s) = \sum_{i=1}^{N^2-1} (|row_i - row_{\text{goal}_i}| + |col_i - col_{\text{goal}_i}|)$$

2.2 Manhattan + Linear Conflict

Ta heurystyka jest rozszerzeniem heurystyki Manhattana, dodając karę za "konflikty liniowe"(dwa kafelki w docelowym wierszu/kolumnie, ale w odwrotnej kolejności). Oferuje silniejsze szacowanie niż sama heurystyka Manhattan.

$$h_{\text{LinearConflict}}(s) = h_{\text{Manhattan}}(s) + 2 \times \text{liczba konfliktów liniowych}$$

3 Generowanie permutacji

3.1 Algorytm Fisher-Yates

Użyłam algorytmu **Fisher-Yates shuffle** do generowania permutacji. Jest to standardowy algorytm, który dla danej sekwencji elementów generuje losową permutację w sposób **równomierny (jednostajny)**. Każda możliwa permutacja ma taką samą szansę wylosowania (zakładając, że pole w prawym dolnym rogu jest zawsze puste i nie uwzględniamy go podczas tasowania wartości łamigłówki - tak jak w treści zadania na liście).

```
---------------------------------------------------------------------------
// ...
// fragment funkcji initializePuzzle
vector<int> base(total);
for (int i = 0; i < total - 1; i++)
    base[i] = i + 1;
base[total - 1] = 0;
initState.tiles = base;
initState.blank = total - 1;
unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
```

```
std::mt19937 generator(seed);
do {
    for (int i = total - 2; i > 0; i--) {
        std::uniform_int_distribution<int> distribution(0, i);
        int j = distribution(generator);
        swap(initState.tiles[i], initState.tiles[j]);
    }
    initState.tiles[total - 1] = 0;
} while (!isSolvable(initState.tiles));
initState.blank = total - 1;
// ...
-------------------------------------------------------------------------
```

3.2 Sprawdzanie rozwiązywalności

Po wygenerowaniu permutacji, sprawdzana jest jej rozwiązywalność za pomocą funkcji ‘isSolvable‘. Układanka o rozmiarze NxN jest rozwiązywalna, jeśli suma liczby inwersji (par kafelków w niewłaściwej kolejności) i numeru wiersza, w którym znajduje się puste pole, spełnia określone kryteria:

- Dla N nieparzystego: układanka jest rozwiązywalna, jeśli liczba inwersji jest parzysta.
- Dla N parzystego: układanka jest rozwiązywalna, jeśli suma liczby inwersji i numeru wiersza pustego pola (liczonego od dołu) jest nieparzysta.

Losowanie jest powtarzane, dopóki nie zostanie znaleziona rozwiązywalna konfiguracja.

```
-------------------------------------------------------------------
bool isSolvable(const std::vector<int> &tiles) {
    int inv = 0;
    int blank_row = 0;
    for (int i = 0; i < tiles.size(); i++) {
        if (tiles[i] == 0) {
            blank_row = (N - 1) - (i / N);
        }
        for (int j = i + 1; j < tiles.size(); ++j) {
            if (tiles[i] != 0 && tiles[j] != 0 && tiles[i] > tiles[j]) {
                inv++;
            }
        }
    }
    if (N % 2 == 1)
        return inv % 2 == 0;
    return (inv + blank_row) % 2 == 1;
}
-------------------------------------------------------------------
```

4 Opis działania algorytmu Bidirectional A*

Algorytm Bidirectional A* jest rozszerzeniem standardowego algorytmu A*, który przeszukuje jednocześnie z dwóch stron: od stanu początkowego do celu (w przód) i od stanu docelowego do początku (wstecz). Przeszukiwanie zatrzymuje się, gdy oba "spotkają się".

4.1 Graf połączeń i przeglądanie stanów

Przestrzeń stanów układanki można traktować jako **graf**, gdzie każdy wierzchołek reprezentuje unikalną konfigurację kafelków, a krawędzie łączą stany, które można osiągnąć za pomocą pojedynczego ruchu pustego pola. Krawędzie mają wagę 1.

Algorytm utrzymuje dwie kolejki priorytetowe (‘openF‘, ‘openB‘) i dwie mapy odwiedzonych stanów (‘visitedF‘, ‘visitedB‘):

- ‘openF‘, ‘openB‘: przechowują węzły do odwiedzenia, sortowane według funkcji $f = g + h$.
- ‘visitedF‘, ‘visitedB‘: mapują odwiedzone stany na parę (koszt g, stan-rodzic), co pozwala na odtworzenie ścieżki i uniknięcie cykli.

W każdej iteracji algorytm wybiera węzeł z kolejki o mniejszej wartości f_{top}. Następnie generuje sąsiadów wybranego stanu, oblicza ich koszt g_2 (koszt dotarcia do tego stanu) oraz h_2 (heurystyka), i dodaje je do odpowiedniej struktury, jeśli znaleziono krótszą ścieżkę lub stan jest nowy.

4.2 Znajdowanie rozwiązania

- Spotkanie obu przeszukiwań następuje, gdy stan rozszerzony z jednej strony (‘s‘) znajduje się w mapie ‘visited‘ z drugiej strony. Wtedy obliczany jest potencjalny całkowity koszt ‘totalCost = g_forward(s) + g_backward(s)‘. Algorytm kontynuuje rozszerzanie, dopóki suma f wartości na szczycie obu kolejek priorytetowych jest mniejsza niż ‘bestCost‘ (najlepszy znaleziony koszt spotkania).
- Po znalezieniu spotkania, ścieżka do rozwiązania jest rekonstruowana poprzez cofanie się od stanu spotkania do stanu początkowego (używając ‘visitedF‘) i od stanu spotkania do stanu docelowego (używając ‘visitedB‘). Następnie te dwie części ścieżki są łączone.

4.3 Złożoność pamięciowa i czasowa

- **Złożoność pamięciowa** jest zależna głównie od rozmiaru map ‘visitedF‘ i ‘visitedB‘ (typu ‘unordered_map‘). Dla układanki $N = 3$, przestrzeń stanów to $9!/2 \approx 180,000$ rozwiązywalnych stanów. Dla $N = 4$, przestrzeń stanów to $16!/2 \approx 10^{13}$ stanów, co sprawia, że przechowywanie wszystkich stanów jest niemożliwe. Bidirectional A* redukuje to, ale nadal może być pamięciożerny.
- **Złożoność czasowa** Bidirectional A* zależy od liczby odwiedzonych stanów, a także od kosztu operacji na kolejkach priorytetowych i mapach hash. Dla M odwiedzonych stanów:
  - **Wstawianie/Pobieranie z kolejki priorytetowej:** $O(\log k)$, gdzie k to rozmiar kolejki (zakładając, że jest zaimplementowana np. jako kopiec binarny).
  - **Wstawianie/Wyszukiwanie w ‘unordered_map‘:** Średnio $O(1)$, w najgorszym przypadku $O(M)$ (przy złych funkcjach hash, ale funkcja FNV-1a minimalizuje to ryzyko).
  - **Generowanie sąsiadów:** Stały czas, $O(1)$ dla układanki $N \times N$ (maksymalnie 4 sąsiadów).

  Całkowita złożoność to około $O(M \log M)$ w średnim przypadku, gdzie M to liczba odwiedzonych stanów. Dzięki Bidirectional A* zazwyczaj złożoność czasowa jest redukowana z $O(b^d)$ do $O(b^{d/2})$, gdzie b to współczynnik rozgałęzienia, a d to głębokość rozwiązania.

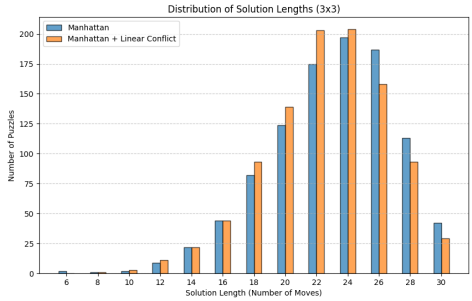
5 Mechanizmy przyspieszające działanie

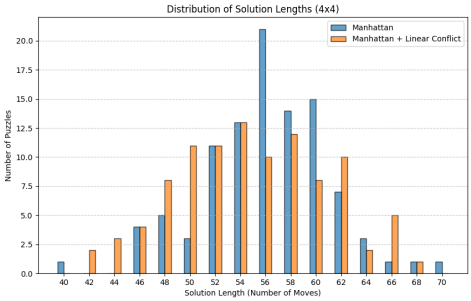
- **Zmniejszenie przestrzeni przeszukiwania**: Głównym mechanizmem przyspieszającym działanie jest algorytm **Bidirectional A***. Zamiast przeszukiwać całą przestrzeń stanów od początku do końca, przeszukuje ją z dwóch stron jednocześnie. Przeszukiwanie dwóch kul o promieniu $d/2$ jest znacznie efektywniejsze niż jednej kuli o promieniu d, redukując liczbę odwiedzonych stanów.
- **Użycie ‘std::unordered_map‘**: Implementacja map hashujących zapewnia średnio czas $O(1)$ dla operacji wstawiania i wyszukiwania, co jest kluczowe dla wydajności przy dużych N. Funkcja hashująca FNV-1a minimalizuje kolizje.

6 Wykresy

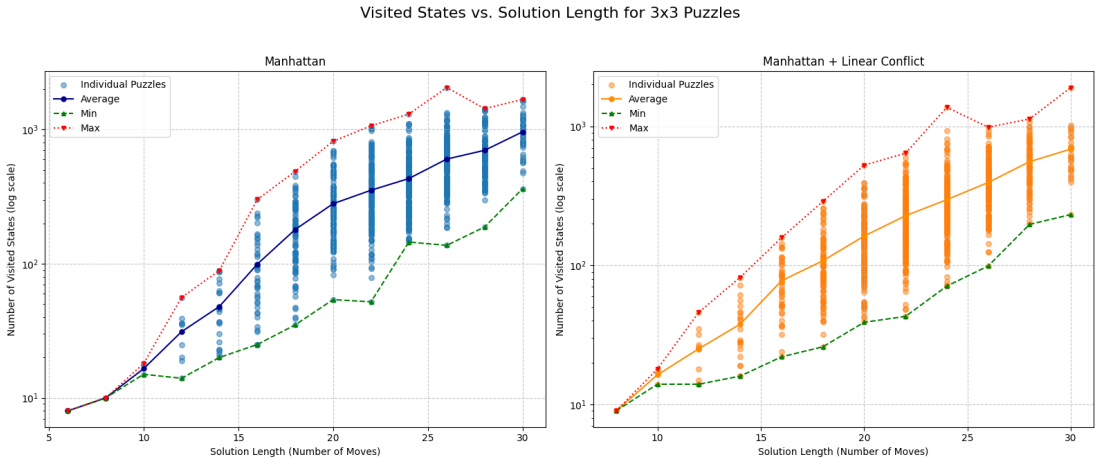
Poniższe wykresy prezentują wyniki eksperymentów dla układanek 3×3 i 4×4, pozwalając na analizę efektywności heurystyk.

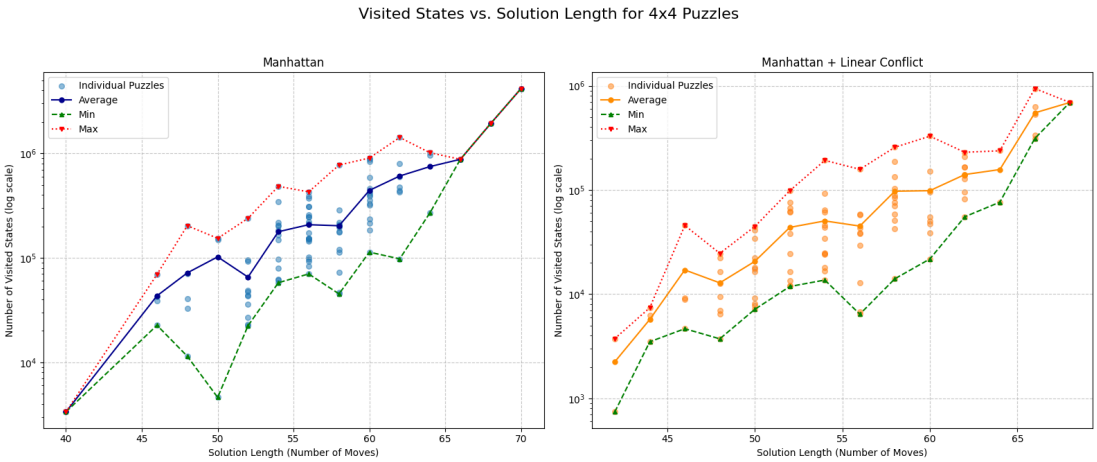
6.1 Rozkład ilościowy długości rozwiązania



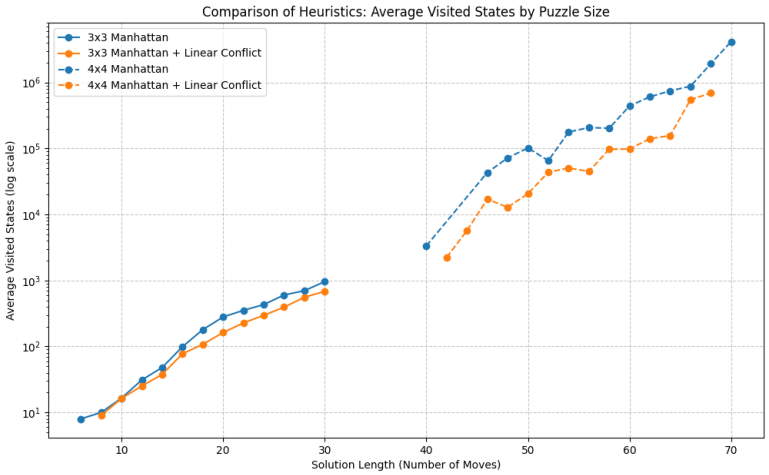


6.2 Liczba odwiedzonych stanów vs. długość rozwiązania

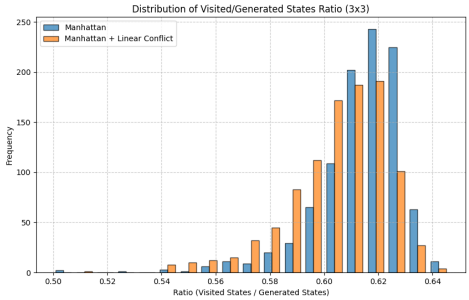


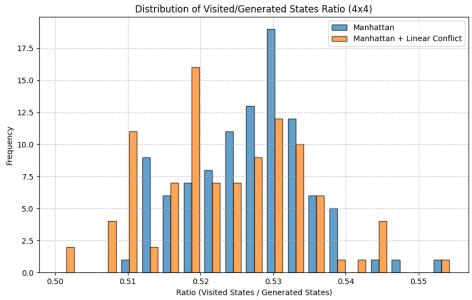


6.3 Porównanie heurystyk: średnia liczba odwiedzonych stanów

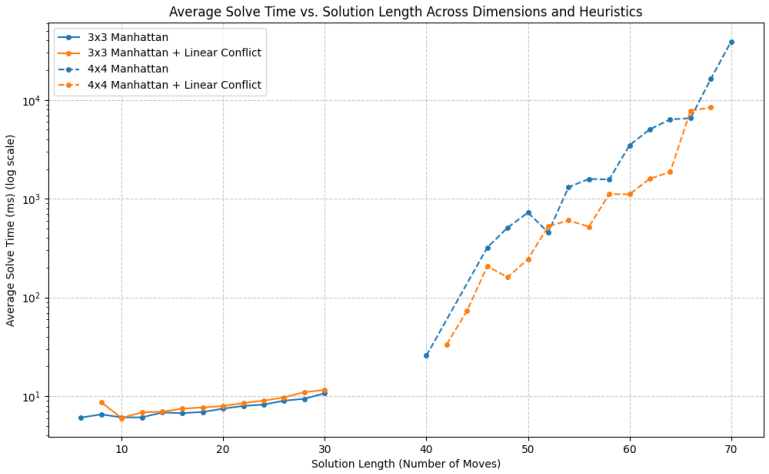


6.4 Stosunek liczby stanów odwiedzonych do wygenerowanych





6.5 Średni czas rozwiązania vs. długość rozwiązania



7 Wnioski

Przeprowadzone eksperymenty potwierdzają, że algorytm Bidirectional A* jest efektywnym narzędziem do rozwiązywania układanki *Piętnastka*.

- Heurystyka **Manhattan + Linear Conflict** jest zdecydowanie bardziej efektywna niż heurystyka **Manhattan**, prowadząc do mniejszej liczby odwiedzonych stanów i krótszego czasu rozwiązania.
- **Bidirectional A** jest skutecznym mechanizmem przyspieszającym, redukującym przestrzeń przeszukiwania.
- Optymalna implementacja struktur danych, takich jak ‘unordered_map‘, ma spore znaczenie dla wydajności.