

# **Naprawianie automatów skończonych z brakującymi stanami i krawędziami**

Repairing Finite Automata  
with Missing Components

**Julia Cygan, Patryk Flama**

Praca inżynierska  
**Promotor:** dr hab. Jakub Michaliszyn

Uniwersytet Wrocławski  
Wydział Matematyki i Informatyki  
Instytut Informatyki

22 stycznia 2026



**Julia Cygan**

.....  
(adres zameldowania)

.....  
(adres korespondencyjny)

PESEL: .....

e-mail: .....

Wydział Matematyki i Informatyki  
stacjonarne studia I stopnia  
kierunek: informatyka  
nr albumu: .....

**Patryk Flama**

.....  
(adres zameldowania)

.....  
(adres korespondencyjny)

PESEL: .....

e-mail: .....

Wydział Matematyki i Informatyki  
stacjonarne studia I stopnia  
kierunek: informatyka  
nr albumu: .....

### **Oświadczenie o autorskim wykonaniu pracy dyplomowej**

Niniejszym oświadczamy, że złożoną do oceny pracę zatytułowaną *Naprawianie automatów skończonych z brakującymi stanami i krawędziami* wykonaliśmy samodzielnie pod kierunkiem promotora, dr Jakuba Michaliszyna. Oświadczamy, że powyższe dane są zgodne ze stanem faktycznym i znane nam są przepisy ustawy z dn. 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (tekst jednolity: Dz. U. z 2006 r. nr 90, poz. 637, z późniejszymi zmianami) oraz że treść pracy dyplomowej przedstawionej do obrony, zawarta na przekazanym nośniku elektronicznym, jest identyczna z jej wersją drukowaną.

Wrocław, 22 stycznia 2026

.....  
(czytelny podpis)

.....  
(czytelny podpis)

# Abstract

Niniejsza praca dotyczy zagadnienia naprawy uszkodzonego deterministycznego automatu skończonego na podstawie próbek pozytywnych i negatywnych. Rozważany automat może zawierać brakujące krawędzie lub stany, co uniemożliwia jego poprawne działanie. W pracy analizowana jest złożoność obliczeniowa problemu, w tym jego przynależność do klasy NP oraz własności parametryzowane w kontekście algorytmów FPT i hierarchii W. Zaproponowano algorytm rozwiązujący rozważany problem, przeprowadzono jego analizę teoretyczną oraz przedstawiono implementację wraz z omówieniem optymalizacji i heurystyk wpływających na czas działania.

---

This thesis addresses the problem of repairing a damaged deterministic finite automaton using positive and negative samples. The considered automaton may contain missing transitions or states, which prevents it from functioning correctly. The work analyzes the computational complexity of the problem, including its membership in the class NP and its parameterized properties in the context of FPT algorithms and the W-hierarchy. An algorithm for solving the problem is proposed, followed by its theoretical analysis, as well as an implementation accompanied by a discussion of optimizations and heuristics affecting the running time.

# Spis treści

<b>1</b>	<b>Wstęp</b>	<b>3</b>
<b>2</b>	<b>Teoria</b>	<b>5</b>
2.1	Definicja problemu, framework . . . . .	5
2.1.1	Definicja problemu z nieokreśloną liczbą stanów . . . . .	5
2.1.2	Redukcja przypadku brakujących stanów . . . . .	6
2.1.3	Definicja problemu z określoną liczbą stanów . . . . .	6
2.2	NP-zupełność . . . . .	6
2.3	Złożoność parametryzowana . . . . .	7
2.3.1	$W[2]$ -trudność . . . . .	8
2.3.2	Przynależność do $W[P]$ . . . . .	10
<b>3</b>	<b>Algorytmy</b>	<b>11</b>
3.1	Podjęście Brute Force . . . . .	11
3.2	Algorytm ze skokami (Jump Tables) . . . . .	12
3.3	Heurystyka naprawy automatu z losowymi restartami . . . . .	14
3.4	Pruning przestrzeni rozwiązań . . . . .	14
<b>4</b>	<b>Implementacja i Eksperymenty</b>	<b>16</b>
4.1	Implementacja . . . . .	16
4.2	Środowisko . . . . .	16
4.3	Sposób testowania . . . . .	16
4.4	Cel . . . . .	17
4.5	Wyniki . . . . .	17
<b>5</b>	<b>Podsumowanie</b>	<b>20</b>
	<b>Bibliografia</b>	<b>20</b>
<b>A</b>	<b>Aneks</b>	<b>22</b>
A.1	Podział pracy . . . . .	22

# Rozdział 1

## Wstęp

Teoria automatów jest dziedziną informatyki teoretycznej, zajmująca się głównie badaniem abstrakcyjnych maszyn wykorzystywanych w celu modelowania obliczeń. Automat to model, który przetwarza dane wejściowe poprzez wykonywanie przejść pomiędzy kolejnymi stanami zgodnie ze zdefiniowanym sposobem reagowania na poszczególne symbole. Automat najczęściej definiowany jest jako graf z oznaczonymi krawędziami i określonymi wierzchołkami początku i końca. Jednym z najważniejszych i najczęściej badanych modeli są automaty skończone, które charakteryzują się skończonym zbiorem stanów.

Jedną z najważniejszych klas automatów skończonych są deterministyczne automaty skończone (ang. *Deterministic Finite Automata*, DFA). W modelu tym dla każdego stanu oraz symbolu alfabetu wejściowego zdefiniowane jest dokładnie jedno przejście do kolejnego stanu. Dzięki tej własności działanie automatu jest jednoznaczne i w pełni przewidywalne, co znacząco upraszcza jego analizę oraz implementację.

W kontekście uczenia automatów wyróżnia się dwa główne podejścia: uczenie aktywne i uczenie pasywne. Uczenie pasywne polega na konstruowaniu modelu automatu wyłącznie na podstawie gotowego zbioru przykładów wejściowych, bez możliwości zadawania dodatkowych pytań czy testowania hipotez w trakcie procesu uczenia. W praktyce oznacza to, że algorytm otrzymuje skończony zbiór słów oznaczonych jako akceptowane lub odrzucane i na tej podstawie próbuje odtworzyć strukturę automatu, który najlepiej odzwierciedla obserwowane zachowanie systemu. Podejście pasywne jest szczególnie użyteczne w sytuacjach, w których brak jest dostępu do „czarnej skrzynki” systemu lub gdy interaktywne testowanie wszystkich możliwych sekwencji wejściowych jest niemożliwe lub kosztowne. Pomimo swojej prostoty, uczenie pasywne wiąże się z szeregiem trudności teoretycznych i praktycznych, w tym ograniczeniem informacji wynikającym z niekompletności danych, możliwością istnienia wielu automatów zgodnych z tym samym zbiorem próbek oraz problemem minimalizacji otrzymanego modelu.

Jednym z fundamentalnych wyników w teorii pasywnego uczenia języków formalnych było wykazanie, że klasa języków regularnych nie jest identyfikowalna w granicy wyłącznie na podstawie pozytywnych przykładów [1]. Istotnie ogranicza to możliwości pasywnego uczenia automatów skończonych bez dodatkowych założeń. Wynik ten miał istotny wpływ na dalszy rozwój wnioskowania gramatyk, wskazując na konieczność wykorzystywania zarówno przykładów pozytywnych, jak i negatywnych, bądź wprowadzania dodatkowych ograniczeń na strukturę danych uczących lub klasę rozważanych automatów.

W ostatnich latach problem pasywnego uczenia deterministycznych automatów skończonych pozostaje przedmiotem intensywnych badań. W jednej z nowszych prac autorzy koncentrują się na formalnej analizie problemu DFA-consistency, czyli określenia,

czy istnieje minimalny deterministyczny automat skończony, który akceptuje wszystkie pozytywne przykłady i odrzuca wszystkie negatywne przykłady dostarczone w zbiorze uczącym. Badając złożoność obliczeniową tego problemu oraz warianty wynikające z różnych ograniczeń na alfabet i strukturę danych uczących. Wykazano, że problem DFA-consistency jest NP-zupełny, nawet w przypadku alfabetów binarnych, co oznacza, że w ogólności nie istnieje znany algorytm wielomianowy rozwiązujący go dla wszystkich instancji [2].

Inne podejście prezentują prace, w których rekonstrukcja deterministycznych automatów skończonych realizowana jest poprzez redukcję problemu uczenia do problemów spełnialności logicznej (SAT). Przykładem takiego rozwiązania jest narzędzie DFAMiner [3], które konstruuje automat pośredni w postaci trójwartościowego automatu skończonego (3DFA), zawierającego stany akceptujące, odrzucające oraz stany typu nieistotne, umożliwiające dokładne rozpoznanie dostarczonych przykładów uczących. Następnie automat ten jest minimalizowany poprzez redukcję do problemu SAT, co pozwala na uzyskanie minimalnego automatu separującego, czyli deterministycznego automatu skończonego o najmniejszej możliwej liczbie stanów, który akceptuje wszystkie przykłady pozytywne i jednocześnie odrzuca wszystkie przykłady negatywne. Tego rodzaju automat nie musi w pełni określać języka docelowego, lecz jedynie rozdzielać (separować) dostarczone zbiory próbek. Przeprowadzone badania empiryczne wskazują, że tego typu podejścia mogą znacząco przewyższać klasyczne metody uczenia pasywnego pod względem efektywności obliczeniowej. Jednocześnie skuteczność metod opartych na redukcji do SAT pozostaje silnie uzależniona od kompletności oraz spójności danych uczących, a w przypadku próbek niepełnych lub sprzecznych liczba potencjalnych modeli rośnie wykładniczo, co istotnie komplikuje proces uczenia.

Pomimo znacznego postępu w dziedzinie pasywnego uczenia DFA, większość istniejących metod zakłada, że automat uczony jest konstruowany od podstaw na podstawie zbioru przykładów. W praktycznych zastosowaniach często spotyka się jednak sytuacje, w których dostępny jest częściowo zdefiniowany automat, zawierający brakujące stany, niepełne przejścia lub fragmentaryczną wiedzę o strukturze systemu. Tego rodzaju przypadki pojawiają się m.in. w inżynierii odwrotnej, analizie dziedziczonych systemów, rekonstrukcji protokołów komunikacyjnych oraz w procesach naprawy modeli formalnych.

Celem niniejszej pracy jest analiza problemu naprawy brakujących deterministycznych automatów skończonych na podstawie skończonego zbioru przykładów pozytywnych i negatywnych. Przez brakujący deterministyczny automat skończony rozumiany jest automat, w którym zbiór stanów oraz część przejść są określone poprawnie, natomiast pozostałe przejścia nie zostały zdefiniowane. Naprawa automatu polega na uzupełnieniu brakujących przejść w taki sposób, aby otrzymany automat był deterministyczny oraz zgodny z dostarczonym zbiorem przykładów. W rozważanym problemie zakłada się, że automat wejściowy jest dany z góry, a zbiór przykładów pozytywnych i negatywnych jest niesprzeczny, tzn. istnieje co najmniej jeden deterministyczny automat skończony, który jest zgodny zarówno z istniejącą strukturą automatu, jak i z dostarczonymi danymi uczącymi.

W pracy skoncentrowano się na teoretycznej analizie złożoności obliczeniowej problemu naprawy deterministycznych automatów skończonych. W szczególności wykazane zostanie, że rozważany problem jest NP-zupełny, a ponadto omówiona zostanie jego trudność w sensie klasy parametryzowanej  $W[2]$ . Oprócz wyników teoretycznych zaprezentowany zostanie algorytm, rozwiązujący ten problem w czasie lepszym niż podejście brute force, wykorzystujący skoki przez zdefiniowane krawędzie.

# Rozdział 2

## Teoria

### 2.1 Definicja problemu, framework

#### Definicja 2.1.1 - Deterministyczny automat skończony (DFA)

Deterministyczny automat skończony (DFA) to krotka  $(Q, \Sigma, \delta, q_\lambda, \mathbb{F}_A, \mathbb{F}_R)$ , gdzie:

- $Q$  to skończony zbiór stanów,
- $\Sigma$  to skończony alfabet wejściowy,
- $\delta : Q \times \Sigma \rightarrow Q$  to funkcja przejścia,
- $q_\lambda \in Q$  to stan początkowy,
- $\mathbb{F}_A \subseteq Q$  to zbiór stanów akceptujących,
- $\mathbb{F}_R \subseteq Q$  to zbiór stanów odrzucających.

#### 2.1.1 Definicja problemu z nieokreśloną liczbą stanów

##### Definicja 2.1.2 - Problem naprawienia częściowego DFA z nieokreśloną liczbą stanów

**Wejście:** częściowy automat deterministyczny  $A = (Q, \Sigma, \delta, q_\lambda, \mathbb{F}_A, \mathbb{F}_R)$ , w którym:

- niektóre krawędzie w automacie mogły zostać usunięte, więc dla niektórych par  $(q, a) \in Q \times \Sigma$  funkcja  $\delta$  nie jest określona,
- niektóre stany  $q$  mogły zostać usunięte z automatu (brakujące stany), więc nie należą one ani do  $\mathbb{F}_A$ , ani do  $\mathbb{F}_R$ ,

oraz zbiory próbek  $S^+ \subseteq \Sigma^*$  (słowa akceptowane) i  $S^- \subseteq \Sigma^*$  (słowa odrzucane).

**Wyjście:** odpowiedź, czy istnieje uzupełnienie brakujących przejść, klasyfikacji stanów i ewentualne dodanie stanów tak, aby otrzymany automat był deterministyczny oraz akceptował wszystkie słowa z  $S^+$  i odrzucał wszystkie słowa z  $S^-$ . W przypadku istnienia, należy podać takie uzupełnienie z najmniejszą liczbą dodatkowych stanów.



## 2.1.2 Redukcja przypadku brakujących stanów

W rozważanej wersji problemu dopuszczamy dodawanie brakujących stanów. Możemy ograniczyć maksymalną liczbę stanów w automacie do liczby stanów w automacie, opartym na drzewie prefiksowym zbudowanym z próbek. Jeżeli automat jest naprawialny, to będzie to automat rozwiązujący problem. Rozmiar takiego drzewa możemy ograniczyć przez  $N = |S^+ \cup S^-| \cdot \max_{w \in S^+ \cup S^-} |w|$ . Możemy więc dla każdej liczby stanów  $n$  w zakresie  $[1, N]$  sprawdzać, czy istnieje naprawa automatu z dokładnie  $n$  stanami; od pewnej wartości  $n$  odpowiedź staje się pozytywna, co pozwala użyć wyszukiwania binarnego bez istotnej zmiany (i tak wykładniczej) złożoności.

Dlatego w dalszej części pracy zakładamy, że liczba stanów jest ustalona i nie rozważamy dodawania nowych.

## 2.1.3 Definicja problemu z określoną liczbą stanów

### Definicja 2.1.3 - Problem naprawienia częściowego DFA

**Wejście:** częściowy automat deterministyczny  $A = (Q, \Sigma, \delta, q_\lambda, \mathbb{F}_\mathbb{A}, \mathbb{F}_\mathbb{R})$ , w którym dla pewnych par  $(q, a) \in Q \times \Sigma$  funkcja  $\delta$  nie jest określona, oraz zbiory próbek  $S^+ \subseteq \Sigma^*$  i  $S^- \subseteq \Sigma^*$ .

**Wyjście:** odpowiedź, czy istnieje uzupełnienie brakujących przejść i klasyfikacji stanów tak, aby otrzymany automat był deterministyczny, akceptował wszystkie słowa z  $S^+$  i odrzucał wszystkie słowa z  $S^-$ . W przypadku istnienia należy podać takie uzupełnienie. Zauważamy, że w tym problemie nie możemy dodawać nowych stanów do automatu.

## 2.2 NP-zupełność

Aby wykazać że problem naprawiania częściowego DFA 2.1.3 jest NP-zupełny, musimy udowodnić, że należy do klasy NP oraz że jest NP-trudny.

### Lemat 2.2.0.1 - Problem naprawienia częściowego DFA należy do klasy NP

*Dowód.* Problem naprawienia częściowego DFA należy do klasy NP, ponieważ dla danej instancji problemu możemy w czasie wielomianowym zweryfikować poprawność podanego rozwiązania.

Weryfikacja polega na sprawdzeniu, czy uzupełniony automat jest deterministyczny oraz czy akceptuje i odrzuca odpowiednie próbki. Sprawdzenie deterministyczności automatu wymaga przejrzenia wszystkich stanów i liter alfabetu, co zajmuje czas  $O(|Q| \cdot |\Sigma|)$ . Sprawdzenie klasyfikacji próbek wymaga przejścia przez każdą próbkę i symulacji jej działania na automacie, co zajmuje czas  $O((|S^+| + |S^-|) \cdot M)$ , gdzie  $M$  to maksymalna długość próbki. Ponieważ oba te kroki można wykonać w czasie wielomianowym względem rozmiaru wejścia, problem naprawienia częściowego DFA należy do klasy NP.  $\square$

### Fakt 2.2.1

Problem *najmniejszego zgodnego automatu* jest NP-zupełny [4]:

### Definicja 2.2.1 - Problem najmniejszego zgodnego automatu

**Wejście:** liczba naturalna  $n \in \mathbb{N}$  oraz dwa zbiory słów nad alfabetem  $\Sigma$ : zbiór słów akceptowanych  $S^+ \subseteq \Sigma^*$  oraz zbiór słów odrzucanych  $S^- \subseteq \Sigma^*$ .

**Wyjście:** odpowiedź, czy istnieje deterministyczny automat skończony (DFA)  $\mathcal{A}$ , z co najwyżej  $n$  stanami, taki że wszystkie słowa z  $S^+$  są akceptowane przez  $\mathcal{A}$ , a wszystkie słowa z  $S^-$  są odrzucane przez  $\mathcal{A}$ .

### Lemat 2.2.0.2 - Problem naprawienia częściowego DFA jest NP-trudny

*Dowód.* Możemy przeprowadzić redukcję z problemu *najmniejszego zgodnego automatu* do naszego problemu *naprawienia częściowego DFA*.

Weźmy instancję problemu *najmniejszego zgodnego automatu* - liczbę naturalną  $n$  oraz zbiory słów  $S^+$  oraz  $S^-$ . Stwórzmy częściowy automat DFA  $A = (Q, \Sigma, \delta, q_\lambda, \mathbb{F}_A, \mathbb{F}_R)$ , gdzie  $|Q| = n$  oraz dla wszystkich stanów  $q \in Q$  oraz liter  $a \in \Sigma$  przejście  $\delta(q, a)$  jest nieokreślone, natomiast zbiory próbek są takie same jak w oryginalnym problemie.

Wtedy odpowiedź na problem *naprawienia częściowego DFA* dla automatu  $A$  oraz próbek  $S^+$  i  $S^-$  jest pozytywna wtedy i tylko wtedy gdy odpowiedź na problem *najmniejszego zgodnego automatu* dla liczby  $n$  oraz próbek  $S^+$  i  $S^-$  jest pozytywna.

□

## 2.3 Złożoność parametryzowana

Dowodzona NP-zupełność problemu naprawienia częściowego automatu dotyczy przypadku, gdy z automatu usunięto wszystkie krawędzie. Jest to skrajna sytuacja. W ramach tej pracy chcieliśmy się skupić na sytuacji, w której liczba brakujących krawędzi jest ograniczona. Zauważmy, że jeżeli w automacie o  $n$  stanach brakuje tylko jednej krawędzi, to do jego naprawy wystarczy sprawdzić  $n$  automatów – po jednym dla każdego możliwego końca tej krawędzi. Tę obserwację można uogólnić do następującego faktu.

### Twierdzenie 2.3.1

Dla każdego ustalonego  $k$ , problem naprawiania częściowego automatu ograniczony do automatów z co najwyżej  $k$  brakującymi krawędziami jest w PTIME.

*Dowód.* Niech dany będzie częściowy automat deterministyczny  $A$  o  $n$  stanach z co najwyżej  $k$  brakującymi krawędziami. Algorytm polega na wyliczeniu wszystkich możliwych uzupełnień automatu poprzez dodanie brakujących krawędzi.

Liczba wszystkich możliwych krawędzi w automacie to co najwyżej  $n \cdot |\Sigma|$ , gdzie  $|\Sigma|$  to rozmiar alfabetu. Każda brakująca krawędź może być skierowana do dowolnego spośród  $n$  stanów, zatem liczba możliwych uzupełnień to co najwyżej  $n^k$ .

Dla każdego z tych uzupełnień sprawdzamy w czasie wielomianowym (zależnym od rozmiaru automatu), czy powstały automat spełnia warunki specyfikacji. Ta weryfikacja wykonywana jest w czasie  $O(n^c)$  dla pewnej stałej  $c$ .

Całkowita złożoność wynosi  $O(n^k \cdot n^c) = O(n^{k+c})$ , co jest wielomianem względem rozmiaru wejścia dla ustalonego  $k$ . Zatem problem należy do PTIME dla każdego ustalonego  $k$ .

□

Złożoność parametryczna (ang. *parameterized complexity*) to framework do analizy problemów obliczeniowych, w których oprócz rozmiaru wejścia istnieje dodatkowy parametr. Zamiast klasyfikacji problemów jedynie jako P lub NP-zupełne, badamy zależność złożoności od tego parametru.

Z Twierdzenia 2.3.1 można wywnioskować, że parametryczna wersja problemu naprawiania częściowego automatu należy do klasy XP, czyli klasy problemów, które można rozwiązać w czasie  $n^{f(k)}$ , gdzie  $f(k)$  jest funkcją tylko od parametru  $k$ .

Nas interesuje dogłębnierze zbadanie parametrycznej złożoności tego problemu, a w szczególności ustalenie, czy problem ten należy do klasy FPT, która jest parametrycznym odpowiednikiem klasy PTIME.

### Definicja 2.3.1 - Klasa FPT

Klasa FPT składa się z problemów, dla których istnieje algorytm rozwiązujący je w czasie  $f(k) \cdot n^c$ , gdzie  $n$  to rozmiar wejścia,  $f$  to dowolna funkcja obliczalna, a  $c$  to stała niezależna od  $k$ .

### Definicja 2.3.2 - Klasy W-hierarchii

W-hierarchia to zbiór klas złożoności, oznaczonych jako  $W[i]$ . Zparametryzowany problem znajduje się w klasie  $W[i]$ , jeśli można go zredukować do problemu *Weighted Circuit Satisfiability* dla obwodów o głębokości ograniczonej do  $i$  [5].

Wierzy się że jeżeli problem jest  $W[1]$ -trudny, to nie należy do klasy FPT.

### Definicja 2.3.3 - Redukcja parametryczna

Niech  $(Q, \kappa)$  oraz  $(Q', \kappa')$  będą zparametryzowanymi problemami decyzyjnymi, gdzie  $\kappa$  i  $\kappa'$  są funkcjami parametru. Mówimy, że  $(Q, \kappa)$  *redukuje się parametrycznie* do  $(Q', \kappa')$  (ozn.  $(Q, \kappa) \leq_{\text{fpt}} (Q', \kappa')$ ), jeżeli istnieje funkcja obliczalna  $R$  oraz funkcje obliczalne  $f, g$ , takie że dla każdego wejścia  $x$  zachodzą warunki:

1.  $x \in Q \iff R(x) \in Q'$ ,
2.  $R(x)$  można obliczyć w czasie  $f(\kappa(x)) \cdot |x|^{O(1)}$ ,
3.  $\kappa'(R(x)) \leq g(\kappa(x))$ .

## 2.3.1 $W[2]$ -trudność

W tym rozdziale pokażemy, że problem naprawienia częściowego automatu jest  $W[2]$ -trudny. Do tego celu zredukujemy następujący problem  $W[2]$  zupełny.

### Definicja 2.3.4 - Problem k-zbioru pokrywającego (k-set cover)

Weźmy rodzinę zbiorów  $\mathcal{F}$  stworzoną na uniwersum  $U$ . Dla podrodziny  $\mathcal{F}' \subset \mathcal{F}$  oraz podzbioru  $U' \subset U$  mówimy, że  $\mathcal{F}'$  pokrywa  $U'$ , jeżeli  $\bigcup_{S \in \mathcal{F}'} S \supseteq U'$ . W problemie k-zbioru pokrywającego dane jest uniwersum  $U$ , rodzina zbiorów  $\mathcal{F}$  stworzona na  $U$  oraz liczba naturalna  $k$ . Celem jest znalezienie podrodziny  $\mathcal{F}' \subseteq \mathcal{F}$ , takiej że  $|\mathcal{F}'| \leq k$  oraz  $\mathcal{F}'$  pokrywa  $U$ .

Problem k-zbioru pokrywającego należy do klasy  $W[2]$ -zupełnych problemów [6].

### Twierdzenie 2.3.2

k-zbiór pokrywający  $\leq_{\text{FPT}}$  naprawienie k-częściowego DFA

*Dowód.* Pokażemy redukcję z problemu k-zbioru pokrywającego do problemu naprawienia częściowego DFA. Dla dowolnego wejścia do problemu k-zbioru pokrywającego konstruujemy automat oraz próbki, będące wejściami do problemu naprawienia częściowego DFA, tak aby rozwiązanie problemu naprawienia częściowego DFA rozwiązywało problem k-zbioru pokrywającego.

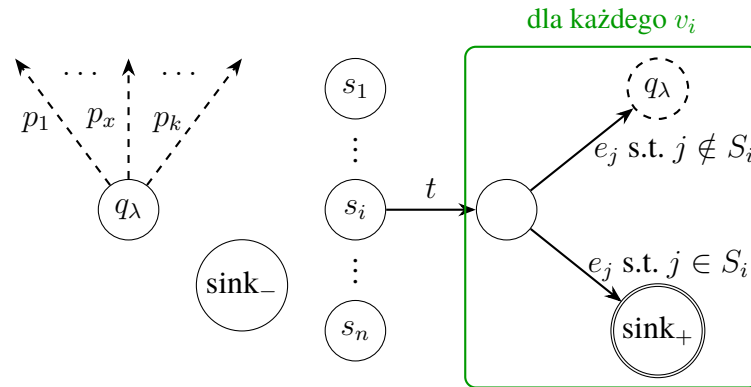
Automat 2.1 z redukcji posiada rozmiar alfabetu zależny od rozmiaru uniwersum  $U$  - można go w analogiczny sposób jak w przypadku redukcji z k-klik zmniejszyć do stałego rozmiaru alfabetu.

Przyjmijmy też, że  $n = |\mathcal{F}|$ .

Idea konstrukcji:

Każdy stan  $s_i$  odpowiada zbiorowi  $S_i \in \mathcal{F}$ . Krawędzie  $e_j$  odpowiadają elementom  $u_j \in U$ . Jeżeli brakująca krawędź  $p_x$  prowadzi do stanu  $s_i$ , to interpretujemy to jako wybranie zbioru  $S_i$  jako  $x$ -tego zbioru w pokryciu.

Sekcja znajdująca się za wierzchołkami  $s_i$  prowadzi do stanu startowego (odrzucającego)  $q_\lambda$ , jeżeli dany zbiór  $S_i$  nie zawiera elementu  $u_j$ . W przeciwnym wypadku prowadzi do stanu akceptującego  $\text{sink}_+$ . Konstrukcja ta reprezentuje elementu z uniwersum  $U$ , które znajdują się w zbiorze  $S_i$ .



Rysunek 2.1. Konstrukcja automatu dla redukcji z  $k$ -zbioru pokrywającego, korzystająca z alfabetu o rozmiarze zależnym od liczby wierzchołków grafu

W tym problemie nie musimy gwarantować wyboru różnych stanów  $s_i$  dla różnych krawędzi  $p_x$ , ponieważ wybór tego samego zbioru wielokrotnie nie ma wpływu na pokrycie uniwersum.

### Gwarancja pokrycia uniwersum

Aby zagwarantować, że wybrane zbiory pokrywają całe uniwersum, dla każdego elementu  $u_j \in U$  tworzymy próbkę przechodzącą do każdego, wybranego przez krawędź  $p_x$ , stanu  $s_i$  oraz sprawdzającą czy dany element  $u_j$  znajduje się w zbiorze  $S_i$ . Jeżeli tak, próbka jest akceptowana, w przeciwnym wypadku odrzucana.

$$\forall j \in \{1..|U|\} [p_1 \ t \ e_j \ p_2 \ t \ e_j \ \dots \ p_k \ t \ e_j] \in S^+ \quad (2.1)$$

Dodatkowo, w analogiczny sposób jak w przypadku redukcji z  $k$ -klik, tworzymy próbki wymuszające prowadzenie brakujących krawędzi do stanów  $s_i$  oraz wybór różnych stanów dla różnych krawędzi  $p_x$ .

### Gwarancja prowadzenia brakujących krawędzi do stanów $s_i$

Poprowadzenie brakującej krawędzi do dowolnego stanu który nie jest  $s_i$  lub  $\text{sink}_+$  doprowadzi do  $\text{sink}_-$  po przejściu następną literą  $t$ .

Aby zapobiec prowadzeniu brakujących krawędzi do stanu  $\text{sink}_+$ , dodajemy próbki odrzucające:

$$\forall x \in \{1..k\} [p_x] \in S^- \quad (2.2)$$

□

### Złożoność konstrukcji

Rozmiar automatu jest wielomianowy w stosunku do rozmiaru danych wejściowych - posiada on  $O(n^2)$  stanów oraz  $n$ -krotnie więcej krawędzi. Czas konstrukcji automatu jest również wielomianowy w stosunku do rozmiaru konstruowanego automatu.

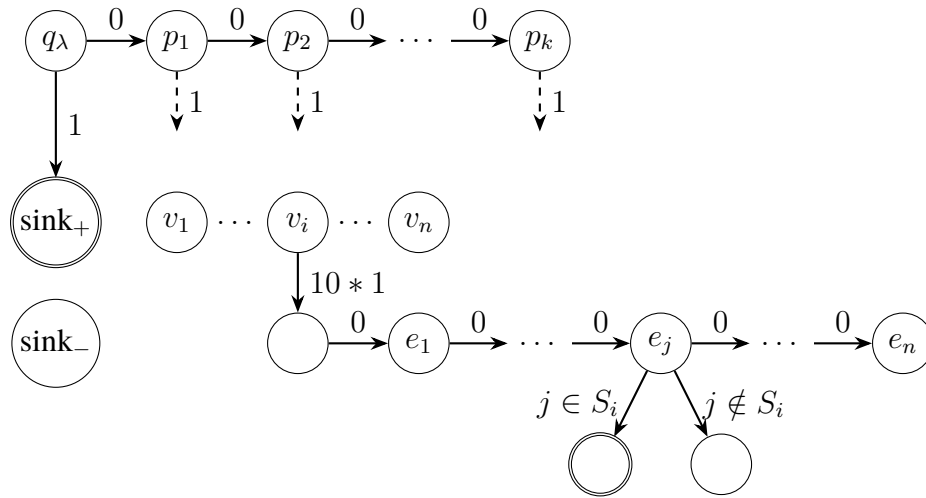
Liczba próbek jest wielomianowa i wynosi  $O(k + |U|)$ , a ich łączna długość wynosi  $O(k)$ .

Powyższy dowód wymaga użycia alfabetu o arności liniowej względem  $k$ . Poniżej pokażemy jednak, że rozmiar alfabetu nie ma znaczenia, dostosowując redukcję do alfabetu rozmiaru 2.

Automat nazwiemy binarnym, jeśli jest nad alfabetem 0, 1.

### Twierdzenie 2.3.3

$k$ -zbiór pokrywający  $\leq_{\text{FPT}}$  naprawianie  $k$ -częściowego automatu binarnego



Rysunek 2.2. Konstrukcja automatu dla redukcji z  $k$ -Set Cover, korzystająca z alfabetu o rozmiarze 2

Dowód. □

### 2.3.2 Przynależność do $W[P]$

Przynależność do  $W[P]$  jeszcze będzie uzupełniona (prawdopodobnie)

# Rozdział 3

## Algorytmy

W niniejszym rozdziale przedstawiono algorytmy służące do naprawy niekompletnego automatu deterministycznego na podstawie zbioru przykładów pozytywnych i negatywnych. Celem jest uzupełnienie brakujących przejść w taki sposób, aby otrzymany automat poprawnie klasyfikował wszystkie próbki wejściowe.

Jako punkt odniesienia omówiono algorytm brute force, który iteruje się po wszystkich możliwych konfiguracjach brakujących przejść i weryfikuje każdą z nich poprzez symulację próbek na automacie. Następnie wprowadzono algorytm ze skokami (jump tables), który przyspiesza etap walidacji poprzez wstępne przetwarzanie próbek i pomijanie fragmentów odpowiadających znanym przejściom.

Dalej przedstawiono podejście heurystyczne oparte na algorytmie hill climbing z losowymi restartami, umożliwiające szybkie znalezienie przybliżonych rozwiązań bez gwarancji optymalności. Na końcu opisano mechanizm pruning, który ogranicza przestrzeń przeszukiwań poprzez odcinanie gałęzi, które nie mogą prowadzić do poprawnego rozwiązania.

Opisane metody różnią się złożonością, czasem działania oraz gwarancjami poprawności i mogą być stosowane zależnie od rozmiaru problemu oraz dostępnych zasobów obliczeniowych.

### 3.1 Podejście Brute Force

Podejście brute force polega na systematycznym sprawdzaniu wszystkich możliwych sposobów uzupełnienia brakujących przejść w automacie. Algorytm generuje kolejne warianty automatu poprzez przypisywanie stanów brakującym przejściom dla poszczególnych symboli alfabetu.

Każdy taki wariant traktowany jest jako osobny kandydat rozwiązania problemu. Dla wygenerowanego automatu algorytm symuluje następnie przetwarzanie wszystkich próbek, przechodząc krok po kroku przez kolejne stany zgodnie z symbolami próbek. Po zakończeniu symulacji sprawdzane jest, czy każda z próbek kończy się w stanie zgodnym z danymi wejściowymi.

Jeżeli dla danego wariantu automatu warunek ten nie jest spełniony, algorytm odrzuca go i przechodzi do analizy kolejnej kombinacji uzupełnień. Proces ten jest powtarzany aż do momentu znalezienia pierwszego wariantu, dla którego wszystkie próbki są poprawnie obsługiwane. W tym momencie algorytm kończy działanie.

### Złożoność obliczeniowa

Czas algorytmu wynosi  $\mathcal{O}(N^k \cdot M \cdot |S|)$ , gdzie  $N$  to liczba stanów,  $k$  to liczba brakujących przejść,  $M$  to maksymalna długość próbki, a  $|S|$  to liczba próbek.

## 3.2 Algorytm ze skokami (Jump Tables)

Algorytm ze skokami stanowi optymalizację symulacji przechodzenia próbką po automacie, z algorytmu Brute Force. Jego głównym celem jest zredukowanie liczby krawędzi, które muszą być przetworzone podczas weryfikacji próbki z automatem.

Możemy zaobserwować, że przechodzenie po *znanych* krawędziach - czyli takich przejściach które dostaliśmy na wejściu - może często prowadzić do redundantnych obliczeń, ponieważ występują one często w automacie, a utworzone z nich ścieżki są jednoznacznie określone dla dowolnej wersji naprawionego automatu.

Dlatego też konstrukcja algorytmu opiera się na stworzeniu struktury, która dla każdego możliwego sufiksu próbek oraz każdego stanu automatu, pozwala na natychmiastowe wyznaczenie stanu docelowego - osiągalnego przy użyciu wyłącznie znanych krawędzi, przeskakując przy tym możliwie najdłuższy fragment podanego sufiksu.

Do algorytmu wprowadzany jest etap wstępnego przetwarzania, gdzie opisana struktura jest wyliczana w postaci tablic skoków. Następnie, podczas walidacji automatu, przechodząc próbką po automacie korzystamy z tablicy skoków, aby przeskoczyć fragmenty składające się ze *znanych* krawędzi.

W efekcie wykonamy jedynie tyle kroków, ile jest *brakujących* krawędzi na ścieżce odpowiadającej danej próbce w automacie.

### Budowa tablic skoków

Tablice skoków budowane są niezależnie dla zbioru przykładów pozytywnych oraz negatywnych. Dla każdej próbki  $w$  o długości maksymalnie  $L$  konstruowana jest tablica  $DP$ , w której wpis  $DP[i][q]$  opisuje efekt przetworzenia najdłuższego możliwego fragmentu próbki  $w$ , zaczynając od pozycji  $i$  w stanie  $q$ , bez użycia brakujących przejść.

---

**Algorytm 1:** Budowa tablic skoków

---

**Wejście:** Automat  $A = (Q, \Sigma, \delta, q_0, F)$ , zbiór próbek  $S$

**Wyjście:** Tablica skoków  $JT$

```
foreach próbka  $w \in S$  do
   $L \leftarrow |w|$ ;
  Utwórz tablicę  $DP[0 \dots L][0 \dots |Q| - 1]$ ;
  foreach stan  $q \in Q$  do
     $DP[L][q] \leftarrow (q, L)$ ;
  for  $i \leftarrow L - 1$  to 0 do
    foreach stan  $q \in Q$  do
      if  $\delta(q, w[i])$  jest nieokreślone then
         $DP[i][q] \leftarrow (q, i)$ ;
      else
         $q' \leftarrow \delta(q, w[i])$ ;
         $DP[i][q] \leftarrow DP[i + 1][q']$ ;
    Dodaj  $DP$  do  $JT$ ;
return  $JT$ 
```

---

**Walidacja automatu z użyciem tablic skoków**

Po skonstruowaniu tablic skoków algorytm wykorzystuje je podczas walidacji każdego kandydata. Zamiast symulować próbki krok po kroku, algorytm wykonuje skoki pomiędzy pozycjami, aż napotka fragment zależny od brakującego przejścia.

---

**Algorytm 2:** Walidacja automatu z wykorzystaniem tablic skoków

---

**Wejście:** Automat  $A$ , próbki  $S$ , tablica skoków  $JT$ , oczekiwany wynik  $b$

**Wyjście:** true jeśli automat jest zgodny z próbkami, false w przeciwnym razie

```
foreach próbka  $w_i \in S$  do
   $q \leftarrow q_0, pos \leftarrow 0$ ;
  while  $pos < |w_i|$  do
     $(q', pos') \leftarrow JT[pos][q]$ ;
    if  $(q', pos') = (q, pos)$  then
      if  $\delta(q, w_i[pos])$  jest nieokreślone then
        Przerwij symulację tej próbki;
       $q \leftarrow \delta(q, w_i[pos])$ ;
       $pos \leftarrow pos + 1$ ;
    else
       $q \leftarrow q', pos \leftarrow pos'$ ;
  if  $(q \in F) \neq b$  then
    return false;
return true
```

---

**Integracja z algorytmem pełnego przeszukiwania**

Algorytm ze skokami nie modyfikuje samej strategii przeszukiwania przestrzeni możliwych uzupełnień brakujących przejść. Zastępuje on jedynie klasyczną procedurę wali-



dacji automatu zoptymalizowaną wersją wykorzystującą tablice skoków. Dzięki temu zachowana zostaje pełna poprawność algorytmu brute force, przy jednoczesnym istotnym zmniejszeniu czasu weryfikacji pojedynczego kandydata w praktyce.

### Analiza wydajności

Czas budowy tablic skoków wynosi  $\mathcal{O}(|S| \cdot M \cdot N)$ , gdzie  $|S|$  to liczba próbek,  $M$  to maksymalna długość próbki, a  $N$  to liczba stanów automatu.

Z założeń wynika, że przy walidacji próbki wykonamy tyle kroków, ile jest brakujących przejść na ścieżce odpowiadającej danej próbce w automacie. W najgorszym przypadku, gdy wszystkie przejścia są brakujące lub co drugie przejście jest brakujące, czas walidacji pozostaje  $\mathcal{O}(M)$ . W praktyce jednak, dla automatu z niewielką liczbą brakujących przejść, czas ten może być znacznie mniejszy.

## 3.3 Heurystyka naprawy automatu z losowymi restartami

Algorytm hill climbing [7] jest algorytmem, który rozpoczyna działanie od pewnego początkowego rozwiązania, a następnie iteracyjnie wprowadza niewielkie modyfikacje, dążąc do poprawy wartości funkcji celu. W każdej iteracji rozważane są rozwiązania sąsiednie, różniące się od aktualnego jedynie pojedynczą zmianą, w tym przypadku: przypisaniem innego stanu docelowego do jednego przejścia automatu. Jeżeli taka modyfikacja prowadzi do poprawy wartości funkcji celu, w tym przypadku: więcej próbek osiąga docelowy stan zgodny z danymi wejściowymi, jest ona akceptowana jako nowe rozwiązanie bieżące.

Metoda ta jest podatna na zatrzymanie się w minimach lokalnych, ponieważ algorytm akceptuje jedynie zmiany prowadzące do poprawy rozwiązania. W celu ograniczenia tego problemu zastosowano losowe restarty algorytmu. Po osiągnięciu minimum lokalnego, w którym nie istnieje żadna poprawiająca modyfikacja, algorytm rozpoczyna działanie od nowej, losowo wygenerowanej konfiguracji niezidentyfikowanych przejść automatu. Proces ten jest powtarzany wielokrotnie, co zwiększa prawdopodobieństwo odnalezienia rozwiązania.

## 3.4 Pruning przestrzeni rozwiązań

W celu ograniczenia liczby analizowanych konfiguracji automatu zastosowano mechanizm odcinania nieperspektywicznych gałęzi przeszukiwań (pruning). Podstawową obserwacją jest fakt, że częściowe ustalenie przejść automatu zawęży zbiór wszystkich możliwych poprawnych konfiguracji.

W trakcie działania algorytmu utrzymywany jest zbiór potencjalnie poprawnych automatów, zgodnych z dotychczas ustalonymi przejściami. Każde przypisanie stanu do kolejnego brakującego przejścia powoduje zawężenie tego zbioru poprzez odrzucenie wszystkich konfiguracji, które nie spełniają nowo wprowadzonego ograniczenia.

Jeżeli na pewnym etapie, przy częściowo ustalonym automacie, zbiór potencjalnych konfiguracji staje się pusty, oznacza to, że żadna kompletna konfiguracja automatu zgodna z danymi wejściowymi nie może powstać w danym kierunku dalszej eksploracji. W takiej sytuacji algorytm przerywa dalsze rozważanie tej gałęzi, niezależnie od tego, że nie wszystkie przejścia zostały jeszcze ustalone.

Zastosowanie pruningu pozwala uniknąć dalszego rozpatrywania konfiguracji, które z góry nie mogą prowadzić do poprawnego rozwiązania, co znacząco redukuje liczbę analizowanych przypadków.

## Rozdział 4

# Implementacja i Eksperymenty

### 4.1 Implementacja

Program został zaimplementowany w C++ 15. Kod jest gotowy do pobrania z publicznego repozytorium na github:

- <https://github.com/PatrykFlama/PracaInz>

Kod jest w folderze programy, w pliku main.cpp można konfigurować liczbę stanów, rozmiar alfabetu, liczbę próbek oraz ich długość, wariancję długości próbek, liczbę brakujących krawędzi, typ automatu oraz liczbę testów i wybrać algorytmy do testowania. Kod można uruchamiać za pomocą Makefile.

### 4.2 Środowisko

Testy zostały wykonane na środowisku o poniższych parametrach:

- System:
- Procesor:
- Architektura:

### 4.3 Sposób testowania

Testy przeprowadzaliśmy na stałych parametrach, jedynie badany parametr jest zmieniany. Automat oraz próbki przygotowywaliśmy w podany sposób:

- generowaliśmy losowy automat pełny (posiadający przejście dla każdego symbolu alfabetu w każdym stanie),
- na podstawie tak wygenerowanego automatu generowaliśmy losowe próbki,
- losowo usuwaliśmy ustaloną liczbę przejść z automatu, które są zawarte w próbce lub próbkach.

Dzięki temu automatycznie wykluczamy wszystkie przypadki niezdefiniowanych przejść w automacie, które mogą zostać poprowadzone w dowolny sposób. Takim sposobem eliminujemy trywialne przypadki, w których dowolny przebieg przejścia algorytmu prowadzi do poprawnego rozwiązania, koncentrując testy na trudniejszych instancjach problemu.

Przy rysowaniu wykresów średni czas wykonania obliczano z wykorzystaniem okna przesuwne. Dane zostały posortowane względem badanego parametru, a następnie dla kolejnych fragmentów danych o stałej liczbie obserwacji wyznaczano średnią arytmetyczną wartości parametru oraz odpowiadających mu czasów wykonania. Okno przesuwne było centrowane, co oznacza, że każda wartość na wykresie reprezentuje średnią obliczoną z obserwacji znajdujących się symetrycznie wokół danego punktu. Zastosowanie okna przesuwne pozwoliło na wygładzenie przebiegu wykresów oraz uwidocznienie ich tendencji.

## 4.4 Cel

Celem przeprowadzonych eksperymentów było zbadanie wydajności algorytmu ze skokami, który stanowi najbardziej obiecujące podejście spośród rozważanych rozwiązań, oraz porównanie jego działania z algorytmem naiwnym. Testy obejmowały ocenę wpływu długości próbek oraz ich liczby na czas wykonania obu algorytmów.

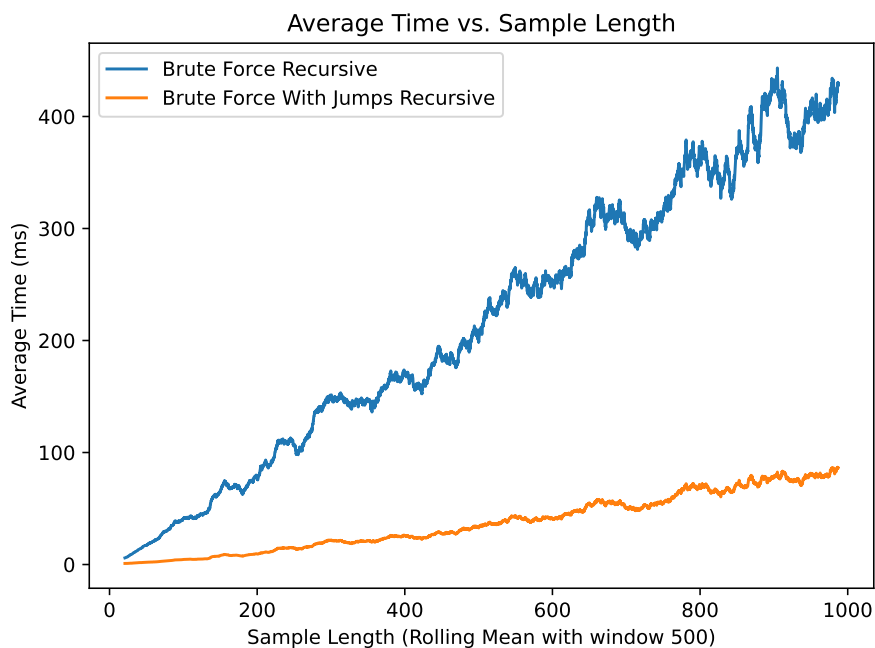
## 4.5 Wyniki

Na wykresie 4.3 obserwowany jest wyraźny wzrost czasu wykonania obu algorytmów wraz ze wzrostem liczby brakujących krawędzi. Algorytm naiwny wykazuje znacznie szybszy przyrost czasu niż wariant wykorzystujący tablice skoków. Algorytm ze skokami charakteryzuje się wolniejszym tempem wzrostu i lepszą skalowalnością względem tego parametru.

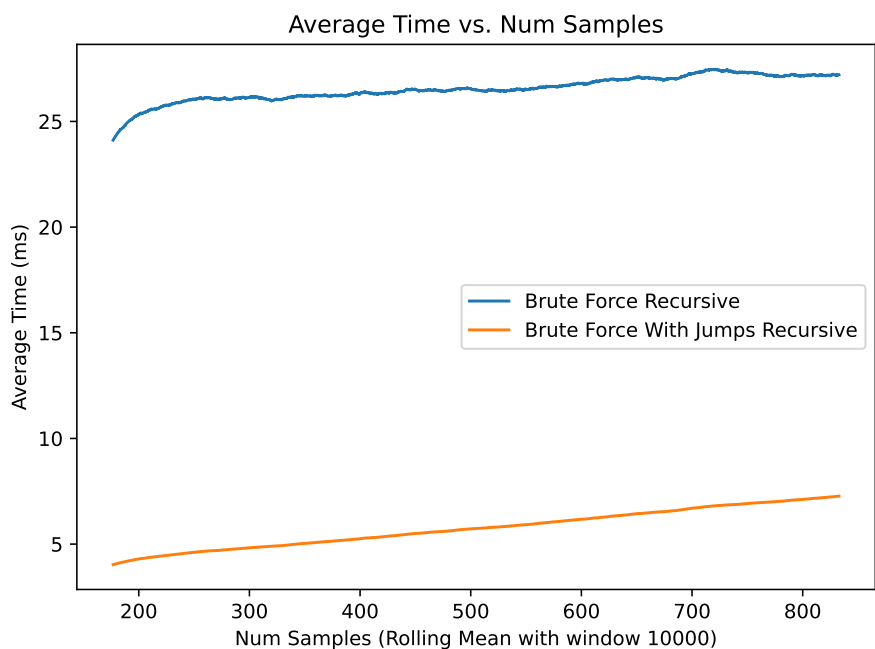
Na wykresie 4.1 widoczny jest liniowy wzrost czasu działania obu algorytmów wraz ze zwiększaniem długości próbek. Algorytm naiwny charakteryzuje się szybkim przyrostem czasu wykonania, osiągając dla największych długości próbek wartości kilkukrotnie wyższe niż algorytm ze skokami. Algorytm ze skokami wykazuje stabilniejszy przebieg oraz wolniejsze tempo wzrostu, co wskazuje na jego lepszą skalowalność względem długości próbek.

Na wykresie 4.2 podobnie obserwowany jest liniowy wzrost czasu działania obu algorytmów wraz ze zwiększaniem ilości próbek, przy czym wpływ liczby próbek jest wyraźnie mniejszy niż wpływ ich długości.

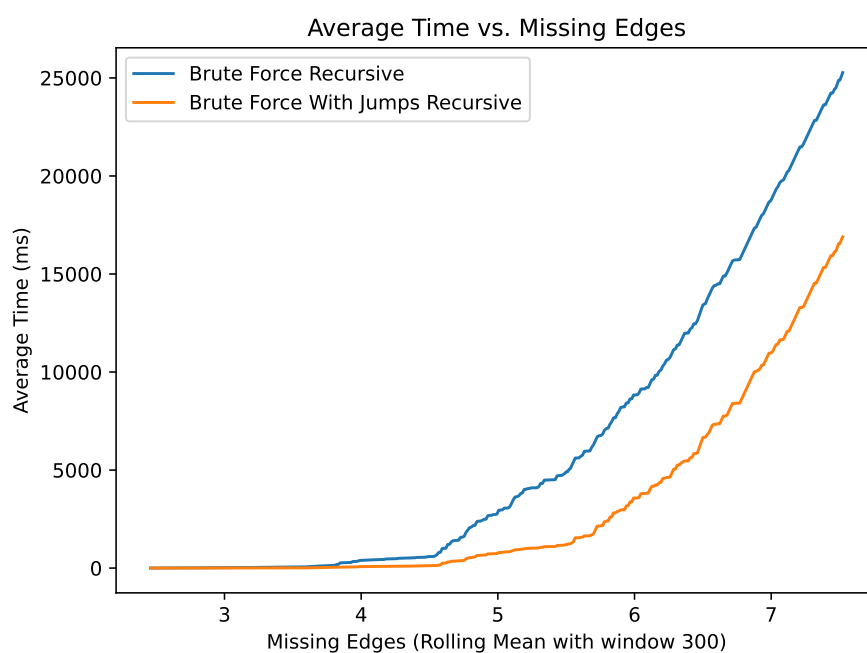
Na podstawie przeprowadzonych testów można stwierdzić, że algorytm ze skokami jest istotnie szybszy od algorytmu naiwnego dla badanych konfiguracji. Uzyskane wyniki potwierdzają jego przewagę wydajnościową zarówno względem długości, jak i liczby próbek. Zaobserwowane zachowanie jest zgodne z oczekiwaniami wynikającymi z charakterystyki analizowanych algorytmów.



*Rysunek 4.1. Średni czas wykonania algorytmu w zależności od długości próbek, liczony na oknie przesuwnym 500. Parametry: 20 stanów, 30 próbek, alfabet 5-symbolowy, 4 brakujące krawędzie, wariancja długości próbek : 0,2, długość próbek z zakresu [30,1000].*



*Rysunek 4.2. Średni czas wykonania algorytmu w zależności od liczby próbek, liczony na oknie przesuwnym 10000. Parametry: 20 stanów, liczba próbek z zakresu [10,1000], alfabet 5-symbolowy, 4 brakujące krawędzie, wariancja długości próbek: 0,2, długość próbek: 30.*



*Rysunek 4.3. Średni czas wykonania algorytmu w zależności od liczby brakujących przejść, liczony na oknie przesuwным 300. Parametry: 20 stanów, 30 próbek, alfabet 5-symbolowy, brakujące krawędzie z zakresu [1,10], wariancja długości próbek: 0,2, długość próbek: 30.*

# Rozdział 5

## Podsumowanie

W niniejszej pracy wykazano, że rozważany problem jest  $W[2]$ -trudny. Wyniki te pozwalają na umiejscowienie badanego zagadnienia w hierarchii klas złożoności parametryzowanej oraz wskazują na istotne ograniczenia możliwości skonstruowania algorytmów efektywnie parametryzowanych. Jednocześnie zasadne wydaje się podjęcie dalszych badań zmierzających do formalnego określenia, czy problem ten jest również  $W[P]$ -trudny, a także czy należy on do klasy  $W[P]$ , co umożliwiłoby pełniejszą charakterystykę jego własności teoretycznych.

Przeprowadzone testy wskazują, że zaproponowany algorytm ze skokami stwarza potencjalne możliwości dalszej optymalizacji, w szczególności w zakresie zapotrzebowania na pamięć.

Dodatkowym kierunkiem optymalizacji jest wykorzystanie struktury trie do wspólnego przetwarzania prefiksów próbek. Podejście to pozwala ograniczyć liczbę powtarzanych przejść automatu w przypadku zbiorów danych zawierających wiele próbek o wspólnych początkach. W praktyce może ono znacząco zmniejszyć liczbę operacji symulacji, zwłaszcza dla dużych zbiorów uczących.

Obiecującym rozszerzeniem algorytmu pełnego przeszukiwania jest również zastosowanie backtrackingu w połączeniu z mechanizmem skoków. Takie podejście umożliwia wczesne odrzucanie nieperspektywicznych częściowych konfiguracji automatu oraz kontynuowanie symulacji próbek od ostatniego poprawnie zdefiniowanego przejścia. W efekcie zmniejsza to zarówno liczbę analizowanych kombinacji uzupełnień, jak i koszt ich weryfikacji.

Istotnym kierunkiem dalszych badań jest również analiza wpływu struktury automatu na czas wykonania algorytmów. W szczególności interesujące wydaje się zbadanie automatu z różną ilością spójnych składowych.

# Bibliografia

- [1] E. M. Gold, „Language Identification in the Limit,” *Information and Control*, s. 447–474, 1967.
- [2] J. Lingg, M. de Oliveira Oliveira i P. Wolf, „Learning from Positive and Negative Examples: New Proof for Binary Alphabets,” *arXiv preprint*, 2022.
- [3] D. Dell’Erba, Y. Li i S. Schewe, „DFAMiner: Mining minimal separating DFAs from labelled samples,” *arXiv preprint*, 2024.
- [4] C. de la Higuera, *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.
- [5] J. Flum i M. Grohe, *Parameterized Complexity Theory*. Springer, 2006.
- [6] M. Cygan, F. V. Fomin, Ł. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk i S. Saurabh, *Parameterized Algorithms*. Springer, 2016.
- [7] S. Russell i P. Norvig, *Artificial Intelligence: A Modern Approach*, 3 wyd. Prentice Hall, 2009.



# **Dodatek A**

## **Aneks**

### **A.1 Podział pracy**

Julia Cygan oraz Patryk Flama wspólnie opracowali problem badawczy. Patryk Flama w przeważającej mierze rozwinął część teoretyczną pracy, w szczególności dowody twierdzeń. Julia Cygan w przeważającej mierze rozwinęła część implementacyjną pracy. Część implementacji miała charakter eksploracyjny i nie została bezpośrednio wykorzystana w treści pracy. Eksperymenty obliczeniowe zostały przeprowadzone wspólnie przez autorów.