Wydział Podstawowych Problemów Techniki
Politechnika Wrocławska

Algorytmy filtrowania w programowaniu ograniczeń

Tomasz Kulik

Praca magisterska napisana
pod kierunkiem
dr Przemysława Kobylańskiego

Politechnika
Wrocławska
Wrocław 2019

Spis treści

Wstęp

Praca swoim zakresem obejmuje wstęp do tematyki globalnych ograniczeń w programowaniu z ograniczeniami oraz propozycję nowego sposobu zapisywania ograniczeń na sekwencjach zmiennych. Celem niniejszej pracy jest zaprojektowanie oraz implementacja nowych algorytmów filtrowania w programowaniu z ograniczeniami oraz porównanie ich efektywności z innymi rozwiązaniami. Implementacja została wykonana w języku C++11 przy użyciu bibliotek z pakietu IBM ILOG CPLEX CP Optimizer. W pracy zostały przedstawione podstawowe pojęcia pozwalające na zrozumienie zagadnienia programowania z ograniczeniami oraz innych wykorzystanych koncepcji. Pierwszy rozdział stanowi krótki wstęp do problematyki programowania z ograniczeniami. Omówiony został w nim mechanizm poszukiwania rozwiązań dopuszczalnych oraz propagacji ograniczeń. Głównym mechanizmem, na którym skupia się praca jest możliwość zdefiniowania ograniczenia w oparciu o skonstruowane przez użytkownika wyrażenie regularne. Rozdział drugi zawiera opis działania algorytmu pozwalającego na wprowadzanie ograniczeń opisanych wyrażeniem regularnym. Dzięki połączeniu zmodyfikowanej metody Thompsona kompilacji *regex* do automatów NFA z algorytmem pozwalającym na przetworzenie sekwencji zmiennych oraz filtrowanie ich dziedzin umożliwiono wprowadzenie wygodniejszej w zdefiniowaniu metody. Nie jest to jednak główna zaleta płynąca z takiego podejścia. Algorytm filtrujący może wykorzystać podane w wyrażeniu informacje dotyczące akceptowanych ciągów i w każdej iteracji odrzucać nieosiągalne wartości z dziedzin zmiennych. W ostatniej części zaprezentowany został sposób użycia nowego ograniczenia w przykładowych modelach. Przedstawiono wyniki analizy wydajności rozwiązanych modeli wykorzystujących wspomniany sposób oraz porównano je z modelami, w których wykorzystano inne predefiniowane ograniczenia dostępne w pakiecie.

Podstawowe pojęcia

Rozdział poświęcony jest między innymi krótkiemu wprowadzeniu w tematykę programowania z ograniczeniami, działania algorytmów filtrujących oraz ich miejsca w modelowaniu. Omówiono również podstawowe zagadnienia z dziedziny języków formalnych jakimi są języki regularne oraz praktyczne sposoby ich opisu. Połączenie powyższych zagadnień pozwala na opis globalnych ograniczeń przy pomocy wyrażeń regularnych na sekwencjach zmiennych o określonych dziedzinach.

2.1 Programowanie z ograniczeniami

Nawiązując do książki "Handbook of Constraints Programming" [3], programowanie z ograniczeniami to paradygmat służący rozwiązywaniu kombinatorycznych problemów przeszukiwania, pojawiających się w wielu dziedzinach techniki m. in. w sztucznej inteligencji, szeroko pojętej informatyce oraz badaniach operacyjnych.

Formalnie problem spełnienia ograniczeń P to taka trójka (X, D, C), gdzie

- X - n-elementowa krotka $(X_1, X_2, ..., X_n)$ zmiennych,
- D - n-elementowa krotka $(D_1, D_2, ..., D_n)$ zbiorów stanowiących dziedziny odpowiadających zmiennych z krotki X, tj. $X_i \in D_i$,
- C - t-elementowa krotka $(C_1, C_2, ..., C_t)$ ograniczeń narzuconych na zmienne z krotki X.

Ograniczenia są to relacje pomiędzy zmiennymi. Innymi słowy ograniczenie C_i jest to para (R_{Si}, S_i), gdzie

- S_i jest krotką złożoną ze zmiennych krotki X,
- R_{Si} jest podzbiorem iloczynu kartezjańskiego dziedzin zmiennych z krotki S_i, tj. $R_{Si} \subseteq (D_1^{Si} \times D_2^{Si} \times ... \times D_k^{Si})$.

Rozwiązaniem problemu P jest n-elementowa krotka $A = (a_1, a_2, ..., a_n)$, w której każde $a_i \in D_i$ oraz wszystkie narzucone ograniczenia zostały spełnione - $(\forall_{(R,S)\in C})\pi_S(A) \in R$, gdzie $\pi_S(A)$ jest to rzut krotki A zawierający zmienne ze zbioru S. Zbiorem $sol(P)$ nazywa się zbiór rozwiązań dopuszczalnych dla problemu P zawierającym wszystkie możliwe rozwiązania spełniające ograniczenia.

W dalszej części rozważany będzie również zbiór $\Omega = D_1 \times D_2 \times ... \times D_n$. Jest to zbiór wszystkich krotek, w których wartość każdej zmiennej przynależy do odpowiedniej dziedziny. Innymi słowy jest to zbiór dopuszczalnych wyników ze względu na dziedziny zmiennych.

Przykład 2.1 ***Problem spełnienia ograniczeń:***

$$\begin{cases} X = (A, B, C), \\ D = (\{2,3\}, \{3,4\}, \{8,16,32\}) \end{cases} \tag{2.1}$$

Powyżej zdefiniowane zostały zmienne oraz ich dziedziny. Następnym krokiem jest określenie ograniczeń na tych zmiennych:

$$\begin{cases} 3A > C \\ 3B > C \\ A \neq B, \end{cases} \tag{2.2}$$

Jedynym dopuszczalnym rozwiązaniem dla powyższego problemu jest trójka: $(A = 3, B = 4, C = 8)$.

2.2 Poszukiwanie rozwiązania

Podstawą działania algorytmów rozwiązujących problemy programowania z ograniczeniami są:

- Przeszukiwanie
- Wnioskowanie

Pierwsze z nich polega kolejnym generowaniu potencjalnych wektorów wynikowych $A' \in \Omega$. Jeśli krotka A' spełnia wszystkie narzucone ograniczenia z C to staje się jednym z potencjalnie wielu rozwiązań danego problemu. Stosuje się różne techniki oraz heurystyki poszukiwania rozwiązania.

Łatwo zauważyć, że przetestowanie wszystkich możliwych krotek wymaga $|\Omega| = |D_1| * |D_2| * ... * |D_n|$ prób. Celem zredukowania mocy zbioru Ω stosuje się metodę wnioskowania, tj. usuwania z dziedzin zmiennych tych wartości, które z pewnością nie mogą zostać wybrane ze względu na narzucone ograniczenia. Taki proces nazywa się propagacją ograniczeń. Również w tej kwestii rozważyć można wiele różnych technik. W ogólności im bardziej algorytm jest w stanie zawęzić dziedziny tym mniej pracy będzie miał algorytm przeszukiwania, co najczęściej przyspiesza znalezienie wyniku.

Połączenie wyżej wspomnianych technik skutkuje szybszym znalezieniem rezultatów. Iteracyjne stosowanie przeszukiwania z nawrotami oraz propagacji ograniczeń zawęża drzewo przeszukiwania, co jest ważnym atutem w przypadku problemów NP-trudnych, do których najczęściej stosuje się technikę programowania z ograniczeniami.

2.3 Propagacja ograniczeń

W przypadku algorytmów przeszukiwania można uciekać się do stosowania różnych heurystycznych metod. Jednak to zawężanie zbioru dopuszczalnych rozwiązań stanowi główną zaletę algorytmu. Podstawową metodą redukcji dziedzin jest propagacja przy użyciu tzw. Network Consistency. Ta metoda pozwala zachować lokalną spójność pomiędzy dziedzinami zmiennych.

Niech zmienne z problemu P będą reprezentowane przez wierzchołki w grafie. Pierwszą z technik jest wprowadzenie spójności wierzchołkowej (ang. Node consistency). Polega to redukcji dziedzin zmiennych tak, by spełniały ograniczenia jednoargumentowe dla nich samych, tj. $D'_i = D_i \cap R_j$, dla każdego R_j będącego ograniczeniem dla zmiennej X_i. Krawędzie pomiędzy wierzchołkami reprezentują ograniczenia binarne pomiędzy parami zmiennych. Dla ograniczenia pomiędzy zmiennymi X_i oraz X_j - R_{ij} należy rozpatrzeć spójność krawędziową - ang. Arc consistency. W tym celu wykluczyć można te wartości dziedzin zmiennych, które nie należą do relacji, tj. wyznaczyć $D'_i = D_i \cap \pi_i(R_{ij})$ oraz analogicznie $D'_j = D_j \cap \pi_j(R_{ij})$. Po zredukowaniu dziedziny danej zmiennej może okazać się, że w grafie powstały kolejne niespójności, więc proces ten należy powtarzać iteracyjnie do momentu osiągnięcia pełnej spójności grafu.

Kolejnym możliwym etapem propagacji ograniczeń jest spójność ścieżek - ang. Path consistency. Analogicznie jak w przypadku spójności krawędziowej można przeanalizować spójność relacji dwóch zmiennych R_{ij} względem dziedziny trzeciej zmiennej X_m. Polega to na wykluczaniu z R_{ij} takich par dopuszczalnych wartości, które nie pozwalają zmiennej X_m przypisać żadnej wartości.

2.4 Globalna spójność

Osiągnięcie ogólnej krawędziowej spójności jest problemem NP-trudnym [3]. Jednak w wielu przypadkach możliwe jest wprowadzenie wyspecjalizowanych algorytmów o niższej złożoności czasowej i/lub pamięciowej. Za sztandarowy przykład można uznać ograniczenie *all_different*, narzucające na zmienne warunek, aby wartości tych zmiennych były parami różne - $\forall_{x'_1, x'_2 \in all\_diff(x_1,...,x_k)} x'_1 \neq x'_2$.

Przykład 2.2 ***Problem all_different:***

$$\begin{cases} X = (A, B, C), \\ D = (\{1,2,3\}, \{1,2,3\}, \{1,2,3\}), \\ \quad \textit{pod warunkiem, że:} \\ \quad \textit{all\_different}(A,\ B,\ C). \end{cases} \tag{2.3}$$

Zbiorem rozwiązań są wszystkie permutacje trójki: $(1, 2, 3)$.

W celu zrozumienia sensu wprowadzania dodatkowych algorytmów dla *all_different* należy rozpatrzeć następny przykład.

Przykład 2.3 ***Problem zmiennych o parami różnych wartościach:***

$$\begin{cases} X = (A, B, C, D), \\ D = (\{1,2,3\}, \{1,2,3\}, \{1,2,3\}, \{1,2,3\}), \\ \quad \textit{pod warunkiem, że:} \\ A \neq B, A \neq C, A \neq D, \\ B \neq C, B \neq D, C \neq D. \end{cases} \tag{2.4}$$

Zbiór rozwiązań jest pusty, ponieważ nie jest możliwe takie wartościowanie, które dla każdej pary zmiennych przyporządkuje różne wartości. Zauważyć można, że spójność wierzchołkowa wraz z krawędziową są zachowane, więc nie nastąpi redukcja dziedzin. Analiza spójności ścieżek również nie wykaże problemów. Algorytm przeszukiwania przetestuje więc cały zbiór Ω *zanim stwierdzi, że problem nie ma rozwiązania. W tym przypadku dopiero sprawdzenie 4-spójności wszystkich kombinacji trzech zmiennych* $< X_i, X_j, X_m >$ *względem czwartej zmiennej* X_k *pozwoliłoby zauważyć sprzeczność, jednak w ogólności badanie n-spójności jest problemem trudnym pod względem złożoności. W przypadku all_different istnieje jednak algorytm o wielomianowym czasie wykonywania, który skutecznie propaguje ograniczenia oraz jest w stanie stwierdzić sprzeczność na etapie samej propagacji [2]. Działa on w oparciu o rozwiązanie problemu na grafie dwudzielnym.*

2.5 Języki regularne

Języki regularne jest to rodzina języków akceptowanych przez automaty skończone (o skończonej liczbie stanów). Gramatykę można opisać w następujący sposób:

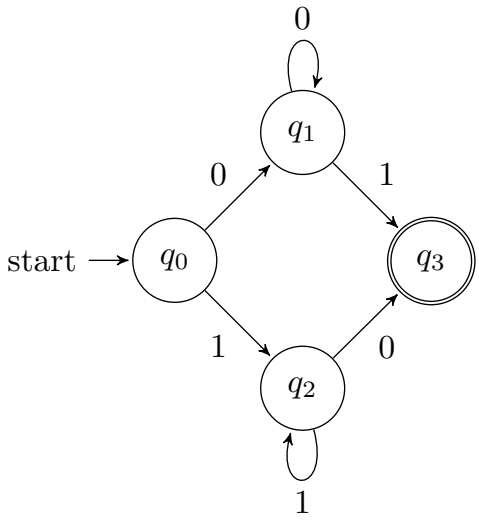
$$\begin{aligned} &A \to a \text{ - symbol } a \text{ jest dowolnym terminalem} \\ &A \to aB \text{ - symbol } a \text{ jest dowolnym terminalem, a } B \text{ nieterminalem} \\ &A \to \epsilon \text{ - gdzie } \epsilon \text{ jest pustym łańcuchem znaków} \end{aligned} \tag{2.5}$$

Innym sposobem przedstawiania gramatyki regularnej są tzw. wyrażenia regularne (ang. regular expressions). Przy pomocy konkatenacji, alternatywy oraz domknięcia Kleene'ego możliwe jest opisanie gramatyki języka regularnego:

- ab - konkatenacja oznacza, że po znaku a musi pojawić się znak b
- $a|b$ - alternatywa wykluczająca, tj. w danym miejscu może pojawić się albo a albo b
- a^* - domknięcie Kleene'ego oznacza 0 lub więcej znaków a

Wyrażenia regularne wzbogacone są również o inne operatory, które w znacznym stopniu ułatwiają zapis wyrażenia, np.

- $a^? \equiv (a|\epsilon)$
- $a^+ \equiv (aa^*)$



Rysunek 2.1: Przykład automatu skończonego, który jest równoważny wyrażeniu regularnemu: (00*1)|(11*0)

- $[a-z0-9] \equiv (a|b|c|...|z|0|1|...|9)$
- $[\hat{}a]$ - dowolny znak poza symbolem a

Automaty skończone można podzielić na dwie klasy - automat deterministyczny (DFA - ang. deterministic finite automaton) oraz automat niedeterministyczny (NFA - ang. non-deterministic finite automaton). Moc obu klas jest równa, co oznacza, że każdą gramatykę regularną można przedstawić zarówno w postaci automatu DFA jak i NFA [4]. Deterministyczny automat skończony jest to piątka $(Q, \Sigma, \delta, s, F)$:

- Q jest skończonym zbiorem stanów
- Σ to alfabet wejściowy
- $\delta : Q \cap \Sigma \leftarrow Q$ jest funkcją przejścia pomiędzy stanami
- $s \in Q$ jest stanem startowym
- $F \subseteq Q$ jest zbiorem stanów akceptujących

Automaty skończone można przedstawić w postaci grafu, w którym wierzchołki oznaczają stany, a krawędzie przejścia pomiędzy stanami (pod warunkiem wystąpienia odpowiedniego symbolu/symboli).

Z formalnego punktu widzenia niedeterministyczny automat skończony NFA z definicji różni się od DFA funkcją przejścia. W przypadku niedeterministycznej wersji występuje $\delta : Q \cap \Sigma \rightarrow 2^Q$, gdzie 2^Q oznacza zbiór potęgowy stanów automatu. Oznacza to, że z danego stanu i symbolu wejściowego można niedeterministycznie przejść do każdego ze stanów określonego podzbioru Q. Istnieje co najmniej kilka metod radzenia sobie ze złożonością obliczeniową automatów NFA, co zostanie poruszone w następnym rozdziale.

Symbol ϵ oznacza pusty łańcuch znaków. Oznacza to możliwość bezwarunkowego skoku do kolejnego stanu bez pobrania symbolu z taśmy wejściowej. Przejście pomiędzy dwoma stanami $S_i \rightarrow S_j$ pod warunkiem wystąpienia ϵ będzie w dalszej części pracy określone przez ϵ-przejście, a ciąg możliwych przejść z danego stanu S_i do S_j oznaczany $S_i \rightarrow S_{k1} \rightarrow S_{k2} \rightarrow ... \rightarrow S_j \equiv S_i \rightarrow^* S_j$ będzie nazywane ϵ-ścieżką.

2.6 Wyrażenia regularne jako opis globalnych ograniczeń

W literaturze można spotkać różne nazwy określające algorytmy propagacji ograniczeń jak np. algorytmy filtrujące, globalne ograniczenia itp. Powstały również zbiory zawierające opisy działania oraz sposoby

implementacji poszczególnych globalnych ograniczeń. Jednym ze sposobów na opisanie rozwiązań dopuszczalnych dla zadanej sekwencji zmiennych jest użycie automatu skończonego [1]. Sekwencje akceptowane przez automat byłyby dopuszczalnymi rozwiązaniami dla danego globalnego ograniczenia, a sekwencja niezaakceptowana oznaczałaby niespełnienie ograniczenia i w konsekwencji natychmiastowy nawrót algorytmu przeszukującego. Jednocześnie na podstawie samego automatu można ograniczać dziedziny zmiennych, co pozwala na skuteczniejszą propagację ograniczenia.

Każdy automat skończony można przedstawić w postaci wyrażenia regularnego oraz każde wyrażenie regularne możne zostać przedstawione w formie automatu skończonego [5]. Zatem dla niektórych globalnych ograniczeń wygodnie byłoby opisać ich działanie w formie wyrażenia regularnego.

Jako dobry przykład może posłużyć problem rozmieszczania rozłącznych odcinków w wektorze zmiennych binarnych. Załóżmy, że $X = (X_1, X_2, ..., X_n)$ oraz, że każda zmienna $X_i \in \{0, 1\}$. Niech w sekwencji X rozmieszczone będzie k odcinków o zadanych długościach $I = (I_1, I_2, ..., I_k)$ wartościowanych jedynkami, a pomiędzy odcinkami zmienne będą wartościowane zerem. Kolejność wystąpienia odcinków jest taka jak w wektorze I.

Przykład 2.4 ***Problem rozmieszczania odcinków:***

$$\begin{cases} X = (X_1, X_2, X_3, X_4, X_5, X_6, X_7), \\ D = (\{0, 1\}, ..., \{0, 1\}), \\ \quad \textit{Niech w wektorze } X \textit{ rozmieszczone będą odcinki } (2, 3). \end{cases} \tag{2.6}$$

Dopuszczalne rozwiązania powyższego problemu to:

$$\begin{gathered} (1, 1, 0, 1, 1, 1, 0) \\ (1, 1, 0, 0, 1, 1, 1) \\ (0, 1, 1, 0, 1, 1, 1) \end{gathered} \tag{2.7}$$

Nawiązując do automatów skończonych, dla powyższego przykładu można skonstruować wyrażenie regularne na sekwencji X opisujące ograniczenia na rozmieszczanie zadanych odcinków: $0^*1^20^+1^20^*$. W ogólności sekwencję $I = (I_1, I_2, ..., I_k)$ można skonstruować na podstawie wzoru $0^*(1^{I_1}0^+)(1^{I_2}0^+)...1^{I_k}0^*$. To oznacza, że dla każdego przykładu powyższe ograniczenie można zapisać w postaci wyrażenia regularnego, a co za tym idzie skonstruować automat NFA. Dalsza część pracy skupiona jest na przetwarzaniu ciągu znaków wyrażenia regularnego na automat NFA oraz zoptymalizowanej metodzie przetwarzania sekwencji zmiennych (ich dziedzin) używając tego automatu. Przedstawiona została również metoda propagacji ograniczeń na podstawie stworzonego automatu.

Analiza problemu

Rozdział zawiera opis ograniczenia filtrującego dziedziny przy pomocy automatu skończonego. Omówiono schemat tworzenia automatu NFA na podstawie wyrażenia regularnego metodą Thompsona. Sposoby opisu globalnych ograniczeń przy pomocy automatów skończonych z licznikiem zostały wprowadzone w różnych implementacjach języka Prolog, co pozwoliło na usprawnienie procesu modelowania [1]. Praca skupia się jednak na automatach skończonych, które nie zawierają liczników, co pozwala na wprowadzenie wygodniejszego interfejsu dla użytkownika (wyrażenia regularne zamiast opis automatu skończonego) oraz większą wydajność.

3.1 Założenia

Algorytm filtrujący ma za zadanie zredukować dziedziny zmiennych celem skrócenia poszukiwania dopuszczalnego rozwiązania. Dla zadanego wyrażenia regularnego R oraz sekwencji zmiennych $(X_1, X_2, ..., X_n)$ proponowany algorytm powinien rozpropagować wśród zadanych zmiennych ograniczenie polegające na akceptacji jedynie takich krotek $(a_1, a_2, ..., a_n)$, które są akceptowane przez automat skończony równoważny z zadanym R. Dla przykładu propagacja w ciągu zmiennych binarnych (X_1, X_2, X_3, X_4) wyrażenia *"0* 1 1 1 0*"* powinna zawęzić dziedziny zmiennych X_2 oraz X_3 do wartości 1. W przypadku stwierdzenia, że dla zmiennych o danych dziedzinach nie jest możliwe wartościowanie akceptowane przez automat, algorytm powinien zgłosić naruszenie ograniczenia.

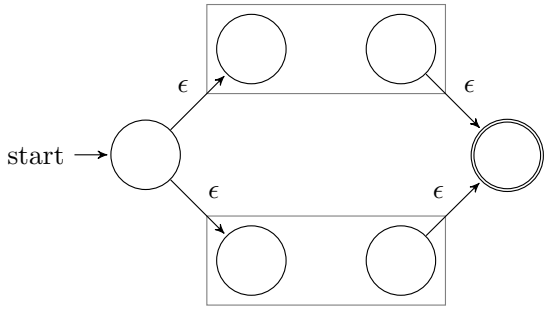
W celu zinterpretowania wyrażenia regularnego na potrzeby propagacji konieczne jest stworzenie na jego podstawie automatu skończonego. W dalszej części rozdziału omówiony został sposób kompilacji wyrażeń regularnych oraz krótka analiza pozytywnych oraz negatywnych cech wybranych metod. Ostatnia część zawiera znane w literaturze algorytmy wykorzystujące automaty niedeterministyczne do testowania zadanego tekstu pod kątem akceptacji przez automat. W kolejnym rozdziale przedstawiono modyfikacje omówionych algorytmów pozwalające na filtrowanie dziedzin zmiennych w problemach programowania z ograniczeniami.

3.2 Kompilacja wyrażenia regularnego

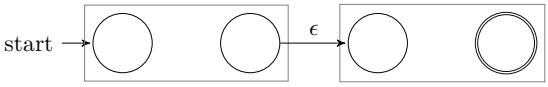
Istnieje co najmniej kilka sposobów na przetłumaczenie wyrażenia regularnego na automat skończony. W przypadku problemów wyszukiwania wzorca w tekście jedną z możliwości jest kompilacja do automatu DFA, ta jednak wymaga asymptotycznie wykładniczej ilości pamięci w stosunku do długości wyrażenia regularnego w najgorszym przypadku. Jest to powód, który czyni tę metodę niepraktyczną pomimo czasu działania zależnego liniowo od długości tekstu do przeszukania.

Drugą metodą jaka się nasuwa jest kompilacja do NFA. Przykładem może być algorytm Thompsona [6], który na podstawie wyrażenia regularnego „składa" automat niedeterministyczny przy pomocy prostych reguł zdefiniowanych dla podstawowych operatorów języków regularnych opisanych w pierwszym rozdziale. Podręcznik "Handbook of Formal Languages vol. 2" [5] wspomina o metodzie Thompsona oraz opisuje drugą wersję pozbawioną tzw. ϵ-przejść pomiędzy stanami. Dla uproszczenia analizy w pracy tej wykorzystano jednak pierwszą z nich.

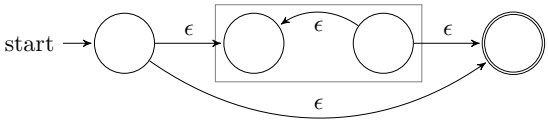
Twierdzenie 3.1 *Metoda Thompsona ma złożoność czasową i pamięciową kompilacji wyrażenia regularnego do NFA ograniczoną przez $O(|r|)$.*



(a) Konstrukcja alternatywy.



(b) Konstrukcja konkatenacji.



(c) Konstrukcja operatora gwiazdki - domknięcia Kleene'ego.

Rysunek 3.1: Powyższa figura przedstawia struktury potrzebne do skonstruowania automatu NFA metodą Thompsona [6].

Pseudokod 3.1: Algorytm znajdujący domknięcie danego stanu, czyli zbiór wszystkich stanów osiągalnych przez ϵ-ścieżki od stanu wejściowego S.

```
1  Function Closure(E, S)
2      R ← S
3      Q ← empty queue
4      foreach p ∈ S do
5          Enque(Q, p)
6      while Q is not empty do
7          p ← Dequeue(Q)
8          foreach state q such that (p, ϵ, q) ∈ E do
9              if q ∉ R then
10                 R ← R + q
11                 Enque(Q, q)
12     return R
```

Pseudokod 3.2: Algorytm symulujący niedeterminizm automatu NFA w czasie wielomianowym i pamięci ograniczonej wielomianem od liczby stanów.

```
1  Function Transitions(E, S, a)
2      R ← empty set
3      foreach p ∈ S do
4          foreach state q such that (p, a, q) ∈ E do
5              R ← R + q
6      return R
```

3.3 Działanie automatu

Choć sama kompilacja działa w czasie liniowym oraz wymaga wielkości pamięci liniowo zależnej od długości wyrażenia regularnego, to kluczowym problemem jest tutaj sam algorytm przetwarzający tekst wejściowy zgodnie z automatem. Metoda przeszukiwania z nawrotami wymaga wykładniczego czasu względem liczby stanów automatu, ponieważ każde miejsce, w którym występuje niedeterministyczne przejście pomiędzy stanami jest przeszukiwane oddzielnie.

Istnieje jednak algorytm, który „symuluje" niedeterminizm występujący w automatach NFA - *RegularExpressionTester* [5]. Zasada jego działania polega na tym, aby utrzymywać zbiór stanów, w którym aktualnie znajduje się automat. Innymi słowy zamiast jednego stanu w danym czasie automat może znajdować się w wielu stanach, aktualizując każdy ze stanów w czasie wczytywania kolejnych znaków z wejścia. Wartość zwrócona przez funkcję *RegularExpressionTester* mówi o tym, czy wejściowy ciąg znaków został zaakceptowany przez automat.

Twierdzenie 3.2 *Metoda RegularExpressionTester z podręcznika „Handbook of Formal Languages" [5] ma złożoność czasową zależną od długości wejścia i liczby stanów $O(n * k)$. Złożoność pamięciowa wynosi $O(k)$.*

Pseudokod 3.3: Algorytm przetwarzający tekst wejściowy zgodnie ze skonstruowanym wcześniej automatem NFA. Wyjściem algorytmu jest wartość $TRUE$ jeśli tekst jest akceptowany przez automat lub $FALSE$ w przeciwnym przypadku.

```
1 Function RegularExpressionTester(x, y)
2     (Q, i, t, E) ← NFA
3     C ← Closure(E, i)
4     foreach letter a from input text do
5         C ← Closure(E, Transitions(E, C, a))
6     return t ∈ C
```

Proponowany algorytm filtrowania

Rozdział zawiera opis algorytmu filtrującego dziedziny zmiennych, z których składa sekwencja wejściowa. Dzięki odpowiedniej modyfikacji metody *RegularExpressionTester* algorytm jest w stanie zebrać informacje dotyczące dopuszczalnych wartości dla każdej zmiennej. W dalszym procesie dane te są wykorzystane do narzucania możliwych wartości w dziedzinach analizowanych zmiennych. Po zaakceptowaniu ciągu wejściowego przez automat, zaczynając od stanu końcowego, algorytm iteruje się od końca sekwencji aż do pierwszej zmiennej kończąc na stanie początkowym. W każdym kroku tej iteracji dla każdej zmiennej zbierane są wartości, dzięki którym możliwe było przejście do stanu końcowego. Wartości te staną się nową dziedziną dla danej zmiennej, a dzięki zapisanym informacjom nt. tego, z którego stanu nastąpiło przejście możliwe jest odtworzenie wszystkich możliwych przejść.

Bazując na literaturze oraz analizie nowego algorytmu zebrane zostały twierdzenia dotyczące teoretycznej złożoności czasowej oraz pamięciowej proponowanej przeze mnie metody. Poddana analizie została również poprawność algorytmu.

4.1 Konstrukcja automatu filtrującego dziedziny

Powyższa metoda może zostać użyta w procesie przetwarzania sekwencji zmiennych o zadanych dziedzinach. Zamiast pojedynczego symbolu ciągu wejściowego a_i można rozważyć zbiór wartości z dziedziny D_i, tj. zaakceptować przejście automatu do kolejnego stanu, jeśli wymagany do przejścia symbol występuje w dziedzinie zmiennej X_i. Głównym celem jest jednak znalezienie możliwości zredukowania dziedzin zmiennych. W takim przypadku ważnymi danymi, jakie algorytm musi uzyskać są dopuszczone przez automat wartości dla każdej zmiennej z sekwencji. Aby to uzyskać konieczne jest śledzenie przejść automatu pomiędzy stanami dla każdej zmiennej wejściowej, więc długość sekwencji staje się kolejnym mnożnikiem w złożoności pamięciowej działania algorytmu.

Pseudokod 4.1: Algorytm bazujący na funkcji *RegularExpressionTester*. Dodanymi przeze mnie krokiem jest zapisywanie historii przejść pomiędzy stanami w czasie procesowania kolejnych zmiennych z sekwencji wejściowej.

```
1  Function evaluateAutomatonWithInputVars(Automaton, Vars)
2      (Q, start, t, E) ← Automaton
3      currentStates ← Closure(E, {start})
4      feasibleValues ← 2D table of size [n][k]
5      foreach variable X_i from input vars sequence do
6          foreach state c ∈ C do
7              foreach state q such that a ∈ D_i AND (p, a, q) ∈ E do
8                  C' ← C' + {q}
9                  feasibleValues[i][c] ← feasibleValues[i][q] + {(p, a)}
10         currentStates ← Closure(E, C')
11     return C, feasibleValues
```

Pseudokod 4.2: Zarys algorytmu propagacji ograniczenia opisanego wyrażeniem regularnym (źródło własne).

```
1 Function RegexFiltering(Automaton, Vars)
2     statesAfterSearching, feasibleValues ← evaluateAutomatonWithInputVars(Automaton, Vars)
3     finishStates ← findFinishStatesAfterSearch(statesAfterSearching, Automaton)
4     if finishStates is empty then
5         violateConstraint()
6     reduceVarsDomains(finishStates, feasibleValues, Vars)
```

Pseudokod 4.3: Algorytm redukujący dziedziny zmiennych na podstawie wyników działania automatu NFA.

```
1  Function reduceVarsDomains(finishStates, feasibleValues, Vars)
2      for i ← size of variables seq. to 1 do
3          nextStates ← empty set
4          goodValues ← empty set
5          foreach state ∈ finishStates do
6              foreach (p, a) ∈ feasibleValues[i][state] do
7                  Insert(nextStates, p)
8                  Insert(goodValues, a)
9          SetDomain(Vars[i], goodValues)
10         Swap(finishStates, nextStates)
```

4.2 Redukcja dziedzin

Jeśli zbiór aktualnych stanów będzie pusty zanim wszystkie zmienne zostaną przetworzone to znaczy, że aktualne dziedziny sekwencji nie są akceptowane przez automat i algorytm propagujący zgłosi naruszenie ograniczenia. Wówczas nastąpi nawrót (ang. backtracking) w algorytmie przeszukiwania. Taka sama sytuacja wydarzy się w przypadku, gdy przeanalizowany zostanie cały ciąg wejściowy i zbiór stanów aktywnych nie będzie zawierał stanu końcowego (lub stanu, który jest połączony ścieżką ϵ-przejść ze stanem końcowym).

Gdy ciąg zmiennych został zaakceptowany przez automat można przystąpić do redukcji dziedzin. Zaczynając od stanu końcowego s_k i ostatniej zmiennej D_n z sekwencji wejściowej dzięki zapisanym informacjom o tym, z którego stanu i jakim symbolem nastąpiło przejście można odtworzyć wszystkie możliwe wartości zaakceptowane przez automat dla D_n. Oznaczając taki zbiór dopuszczalnych wartości przez D'_n można z całą pewnością uznać ten zbiór jako nową dziedzinę dla zmiennej X_n. Przetwarzając kolejne stany i kolejne zmienne możliwa jest redukcja dziedzin dla każdej z nich (jeśli zbiór D'_i jest różny od D_i).

W czasie przeszukiwania algorytm propagujący ograniczenie zostaje uruchomiony za każdym razem, gdy dziedziny zmiennych mu podlegających zostają zmodyfikowane na przykład przez działanie propagacji innych ograniczeń bądź przez ustalenie wartościowania zmiennej (które de facto jest zredukowaniem dziedziny zmiennej do jednej wartości).

Algorytm 4.2 korzysta ze zdefiniowanej wcześniej funkcji *evaluateAutomatonWithInputVars* oraz dwóch dodatkowych *findFinishStatesAfterSearch* i *reduceVarsDomains*. Pierwsza z nich filtruje stany pod kątem tego, czy w ich domknięciach znajduje się stan końcowy. Jeśli nie ma takiego stanu to oznacza, że ograniczenie nie zostało spełnione. Druga z wymienionych funkcji jest bardziej warta uwagi, ponieważ to w niej następuje właściwa redukcja dziedzin zmiennych.

4.3 Poprawność algorytmu

Algorytm *RegexFiltering* wyklucza te wartości z dziedzin sekwencji zmiennych, które nie będą mogły być wykorzystane do poprawnego wartościowania zapewniającego akceptację ciągu przez automat.

Twierdzenie 4.1 *Proponowany algorytm filtrujący RegexFiltering zawsze się zatrzymuje.*

Dowód. Procedura *RegexFiltering* (patrz kod 4.2) wywołuje sekwencyjnie trzy funkcje, którym należy się przyjrzeć. Funkcja *evaluateAutomatonWithInputVars* zawiera trzy zagnieżdżone pętle *for* o ściśle określonych warunkach zakończenia, które w czasie działania tej funkcji są niezmienne. Kolejnym krokiem jest funkcja *findFinishStatesAfterSearch*, która jedynie wyszukuje stany końcowe w domknięciu wszystkich stanów, w jakich zatrzymał się automat. Ostatnia *reduceVarsDomains* podobnie jak pierwsza z omawianych zawiera zagnieżdżone pętle *for*, których warunki zakończenia są niezależne od jej działania i są zdefiniowane w poprawny sposób. ♦

Twierdzenie 4.2 *Proponowany algorytm filtrujący RegexFiltering nigdy nie usunie dopuszczalnego rozwiązania.*

Dowód. Wynika to z samej zasady działania algorytmu. Po przeanalizowaniu całego wejścia przez automat następuje przetwarzanie wypełnionej struktury *feasibleValues* zaczynając od stanu końcowego aż do stanu początkowego. W ten sposób algorytm będzie poruszał się jedynie po ścieżkach zaakceptowanych przez automat, nie omijając żadnych dopuszczalnych wartości. ♦

Z tego samego twierdzenia wynika, że *RegexFiltering* zawsze poprawnie redukuje dziedziny - każde możliwe dopuszczalne rozwiązanie w czasie każdej iteracji znajduje się w przefiltrowanym zbiorze dziedzin $D'_1 \times ... \times D'_n$.

4.4 Złożoność obliczeniowa algorytmu

Twierdzenie 4.3 *Złożoność czasowa algorytmu evaluateAutomatonWithInputVars wynosi $O(n * k^2)$, a złożoność pamięciowa $O(n * k)$.*

Dowód. Pod względem złożoności czasowej, różnicą pomiędzy *evaluateAutomatonWithInputVars* a metodą *RegularExpressionTester* jest zapisywanie historii przejść pomiędzy stanami do tablicy *feasibleValues*. Ponieważ zapis ten odbywa się w czasie $O(1)$ jest on zaniedbywalny w kontekście asymptotycznej złożoności. Jednakże, przez wprowadzenie tzw. ϵ-przejść (patrz rozdział 1.) w automacie, w pesymistycznym przypadku w czasie procesowania każdego stanu S_i może okazać się, że konieczne jest przetworzenie dodatkowo $k - i$ stanów osiągalnych przez ϵ-ścieżki. To oznacza, że dla k stanów algorytm może przeanalizować dla danej zmiennej X_j:

$$\sum_{i=1}^{k} (k - i) = \frac{1}{2}(k - 1)k \in O(k^2) \tag{4.1}$$

Z tego wynika, że przetworzenie całego ciągu wejściowego kosztuje:

$$\sum_{j=1}^{n} \sum_{i=1}^{k} (k - i) \in O(n * k^2) \tag{4.2}$$

Zwiększeniu względem *RegularExpressionTester* podlega też złożoność pamięciowa, ponieważ algorytm ten wymaga tablicy wielkości $n*k$ wektorów par (p, a), gdzie p oznacza stan z którego nastąpiło przejście, a a symbol przejścia. Każdy wektor par może w danym przebiegu algorytmu przechować maksymalnie $l = k * \alpha$, gdzie α jest stałą oznaczającą maksymalną liczbę przejść, jaka może wyjść z jednego stanu. Liczba par jaka może zostać dodana do $feasibleValues[i]$ w czasie przetwarzania zmiennej X_i to również $k * \alpha$, a więc zamortyzowany koszt pamięciowy jednego wektora par dla X_i wynosi $l' = \frac{k*\alpha}{k} = \alpha$, a więc $O(1)$. To oznacza, że wielkości wektorów par nie wpływają na asymptotyczną złożoność pamięciową. ♦

Twierdzenie 4.4 *Złożoność czasowa algorytmu reduceVarsDomains wynosi $O(n * k)$.*

Dowód. Algorytm 4.3 iteracyjnie przechodzi po wszystkich zmiennych ciągu wejściowego oraz po zbiorze aktualnych stanów dodając do zbioru następnych stanów kolejne wpisy, których jest maksymalnie $O(k)$. Zakładając, że implementacja zbioru pozwala na dodanie wpisu w zaniedbywalnym czasie to po złożeniu tych funkcji wynikiem jest $O(n * k)$.

♦

Korzystając z powyższych twierdzeń można wyprowadzić wniosek:

Twierdzenie 4.5 *Złożoność czasowa algorytmu RegexFiltering wynosi* $O(n * k^2)$.

Dowód. Złożoność algorytmu *RegexFiltering* można uzyskać przez zsumowanie złożoności poszczególnych metod składowych, tj. (patrz algorytmy 4.3 i 4.4) oraz *findFinishStatesAfterSearch*. W wyniku daje to $O(n * k^2) + O(n * k) + O(k) = O(n * k^2)$. ♦

Implementacja

Rozdział jest opisem wykonanej implementacji do ograniczenia *RegexConstraint*. Zawarte zostały ważniejsze części kodu kompilującego wyrażenie regularne oraz algorytmu analizującego sekwencję zmiennych pod kątem wygenerowanego automatu. Implementacja została wykonana w języku C++11 z użyciem algorytmów zawartych w bibliotece standardowej oraz pakietu IBM ILOG CPLEX CP Optimizer dla języka C++. Równoważną implementację można wykonać dla języka Java (ze względu na wsparcie ze strony pakietu), jednak wykracza to poza zakres tej pracy.

5.1 Struktury danych

Ważną strukturą, która reprezentuje przejście pomiędzy dwoma stanami jest *Transition* (patrz kod 5.1). Przechowuje ona wartość akceptowaną przez daną relację oraz stan, do którego automat powinien przejść. Dodatkową informacją jest pole *isEmptyString*, które mówi o tym, czy dane przejście ma być tzw. $\epsilon - transition$, czyli przejściem bezwarunkowym. Takie przejście nie pobiera kolejnego znaku z ciągu wejściowego, zamiast tego odbywa się pobierając tzw. pusty łańcuch znaków. Intuicyjnie jeśli automat znajduje się w stanie, w którym są $\epsilon - transition$ to może on niedeterministycznie przełączyć w stany, na które te przejścia pozwalają.

W zaproponowanej implementacji strukturą reprezentującą sam stan jest wektor przejść stanowych. Kolejne indeksy tego wektora oznaczają kolejne możliwe wyjścia z tego stanu.

5.2 Główna struktura ograniczenia

Zaimplementowana klasa *RegexConstraintI* jest strukturą przechowującą wektor wygenerowanych stanów, wektor zmiennych z sekwencji wejściowej oraz wektor wartości dopuszczalnych umieszczonego tu ze względu na optymalizację (raz przygotowana struktura, aby algorytm nie musiał alokować jej za każdym krokiem propagacji). Wartym uwagi konstruktorem jest ten, który przyjmuje jako argumenty wyrażenie regularne oraz wektor zmiennych. W czasie jego działania następuje konstrukcja automatu NFA, który będzie później

Kod źródłowy 5.1: Struktury wykorzystywane w czasie konstruowania automatu NFA oraz propagacji ograniczenia.

```
/*
 * Structures used by RegexConstraint
 */
typedef unsigned int StateNumber;
struct Transition
{
    int value;
    StateNumber newState;
    bool isEmptyString;
};
typedef std::vector<Transition> State;
typedef std::vector<State> StatesVector;
```

wielokrotnie używany w procesie filtrowania dziedzin. Działanie głównych funkcji/metod zawartych w tej klasie zostało opisane w rozdziale 3.

5.3 Parsowanie wyrażeń regularnych

Tłumaczenie wyrażeń regularnych na automaty skończone służy do wstępnego przygotowania danych dla algorytmu filtrującego. W literaturze opisano co najmniej kilka metod, m.in. metodę Thompsona. W zaproponowanej implementacji użyta została wspomniana metoda (jej interpretacja). Istnieją metody, które lepiej dysponują zasobami w postaci czasu działania oraz pamięci. Pozwala to na dalszy rozwój oraz optymalizację proponowanego algorytmu.

Funkcja *parseRegex* (patrz kod 5.3) służy do tłumaczenia wyrażeń regularnych na automaty skończone NFA. Jest to interpretacja metody Thompsona omówionej w poprzednich rozdziałach. Ze względu na bezkontekstowość gramatyki wyrażeń regularnych spowodowanej dodaniem zagnieżdżonych nawiasów należało wprowadzić rekurencyjne odwołania w przypadku napotkaniu symbolu „(". W celu umożliwienia wprowadzania liczb większych niż jednocyfrowe, gramatyka zakłada, że kolejne liczby powinny być oddzielone od siebie znakiem innym niż cyfra. Dla przykładu w przypadku, gdy użytkownik dla zmiennych X_1, X_2, X_3 zechce narzucić ograniczenie, aby $X_1 = 10$, $X_2 = (22)^*$ i $X_3 = 30$, wystarczy określić wyrażenie: 10 22*30.

Wspomniana funkcja iteruje się po kolejnych symbolach z zadanego łańcucha znaków, zaczynając od danego indeksu (jako pierwsze wywołanie metoda zaczyna analizę od początku ciągu, czyli indeksu równemu 0). Główną częścią tej metody jest instrukcja *switch*, która w przypadku napotkania danego symbolu odpowiednio go interpretuje dodając nowy stan, bądź nowe przejścia do istniejących stanów. W przypadku wystąpienia nawiasów funkcja odpowiednio wykonuje rekurencyjnie samą siebie, gdy napotka nawias otwierający oraz powraca do poprzedniego wywołania napotykając nawias zamykający. Za każdym takim razem funkcja dodaje do automatu stan oznaczający przejście po wszystkich symbolach zawartych wewnątrz nawiasu oraz zwraca ostatni przetwarzany indeks, aby funkcja, która ją wywołała mogła kontynuować przetwarzanie od tego miejsca.

Gdy funkcja napotka znak gwiazdki $*$ (domknięcie Kleene'ego) to stworzy w automacie pętlę przez dodanie ϵ-przejścia do początku ostatniego bloku. Znak ? powoduje akceptację poprzedniego symbolu/bloku lub bezwarunkowe pominięcie tego symbolu. W przypadku, gdy analizowanym znakiem będzie | (alternatywa), algorytm zachowa się różnie w zależności od tego, czy kolejnym elementem będzie symbol czy blok zagnieżdżony pomiędzy nawiasami.

Kod źródłowy 5.2: Główna klasa zawierająca implementację ograniczenia *RegexConstraint*.

```
/**
*  Class used by IBM CP solver's engine.
*/
class RegexConstraintI : public IloPropagatorI
{
public:

    /*
    * Constructor of regex constraint. It compiles regex to NFA and prepares feasibleValues
        vector.
    */
    RegexConstraintI(std::vector<IloIntVar> vars, std::string regex);

    /*
    * Constructor used by IBM CP engine.
    */
    RegexConstraintI(StatesVector states,
                     std::vector<StatesVector> rect,
                     std::vector<IloIntVar> vars);

    /*
    * Function used by IBM CP engine
    */
    IloPropagatorI* makeClone(IloEnv env) const override;

    /*
    * Method that is called during constraint propagation. The result of this method's work
        are reduced domains.
    */
    void execute() override;

private:
    void resetFeasibleValuesVectors();
    void addState(State&& state);
    int parseRegex(const std::string& str, uint index);
    std::set<StateNumber> evaluateAutomatonWithInputVars();
    std::set<StateNumber> findFinishStatesAfterSearch(std::set<StateNumber>&
        statesAfterSearching);
    void reduceVarsDomains(std::set<StateNumber>& finishStates);

    /*
    * A vector used to store states created during regex compilation.
    */
    StatesVector states;

    /*
    * A vector of feasible values filled during automaton evaluation. For every state and
        every input variable
    * the algorithm allocates a vector of states and values from wich given state was
        reached.
    */
    std::vector<StatesVector> feasibleValues;

    /*
    * A vector of IBM CP integer variables.
    */
    std::vector<IloIntVar> vars;
};
```

Kod źródłowy 5.3: Metoda kompilująca wyrażenie regularne do automatu NFA

```
int RegexConstraintI::parseRegex(const std::string& str, uint index)
{
    uint prelastState = states.size()-1;
    int currentNumber = 0;
    for(uint i=index; i<str.length(); i++)
        switch(str[i])
        {
            case '(':
                prelastState = states.size();
                i = parseRegex(str, i+1);
            break;
            case ')':
                addState({Transition {-1, (StateNumber) states.size()+1, true}});
                return i;
            break;
            case '*':
                states[states.size()-1][0].newState = prelastState;
                states[prelastState].push_back(Transition {-1, (StateNumber) states.size(),
                    true});
            break;
            case '?':
                states[prelastState].push_back(Transition {-1, (StateNumber) states.size(),
                    true});
            break;
            case '|':
                while(i<str.length()-1 and str[i+1] == ' ')
                    i++;
                if(str[i+1] != '(')
                {
                    states[prelastState].push_back(Transition {-1, (StateNumber) states.size
                        ()+1, true});
                    addState({Transition {-1, (StateNumber) states.size()+2, true}});
                }
                else
                {
                    states[prelastState].push_back(Transition {-1, (StateNumber) states.size
                        ()+1, true});
                    addState({});
                    prelastState = states.size()-1;
                    i = parseRegex(str, i+2);
                    states[prelastState].push_back(Transition {-1, (StateNumber) states.size
                        (), true});
                }
            break;
            default:
                if(str[i] >= '0' and str[i] <= '9')
                {
                    currentNumber = currentNumber*10 + (str[i] - '0');
                    if(i>=str.length()-1 or str[i+1] < '0' or str[i+1] > '9' )
                    {
                        prelastState = states.size();
                        addState({ Transition {currentNumber, (StateNumber) states.size()+1,
                             false} });
                        currentNumber = 0;
                    }
                }
        }
    return str.length();
}
```

Eksperymenty obliczeniowe

Przedstawiony w rozdziale pierwszym przykład ograniczenia opisanego wzorem wyrażenia regularnego może posłużyć np. do rozwiązywania tzw. *Nonogramów*. Jest to również dobry przykład na zaprezentowanie wydajności działania tego algorytmu filtrującego porównując wyniki z modeli korzystających jedynie z dostępnych w pakiecie ograniczeń. W fazie testowej algorytmu dokonano również testów poprawności algorytmu oraz sprawdzono rzeczywistą złożoność czasową względem teoretycznej.

6.1 Propagacja na jednym wektorze

Podstawowymi testami wydajnościowymi dostępnymi w plikach testowych są:

- Zestaw badający złożoność czasową pod kątem rosnącej długości wektora wejściowego.
- Zestaw badający złożoność czasową pod kątem rosnącej złożoności wyrażenia regularnego.

Wyniki zostały zobrazowane na wykresach 6.1 oraz 6.2. Na pierwszym wykresie widać kształt funkcji ograniczonej przez prostą. Można to zinterpretować jako liniowy wzrost złożoności czasowej względem długości wektora zmiennych wejściowych. Pokrywa się to z dowodem 4.5 przeprowadzonym w rozdziale 4. Drugi z wykresów przedstawia funkcję przechylającą się łukiem ku górze, co oznacza ponad liniowy stosunek czasu propagacji do rosnącej złożoności samego wyrażenia regularnego. Przez **złożoność** rozumie się liczbę stanów automatu NFA wygenerowanego na podstawie wyrażenia oraz liczbę ϵ-przejść, jakie w nim występują. W przykładowym teście użyto generatora wyrażeń regularnych bazującego na wzorze:

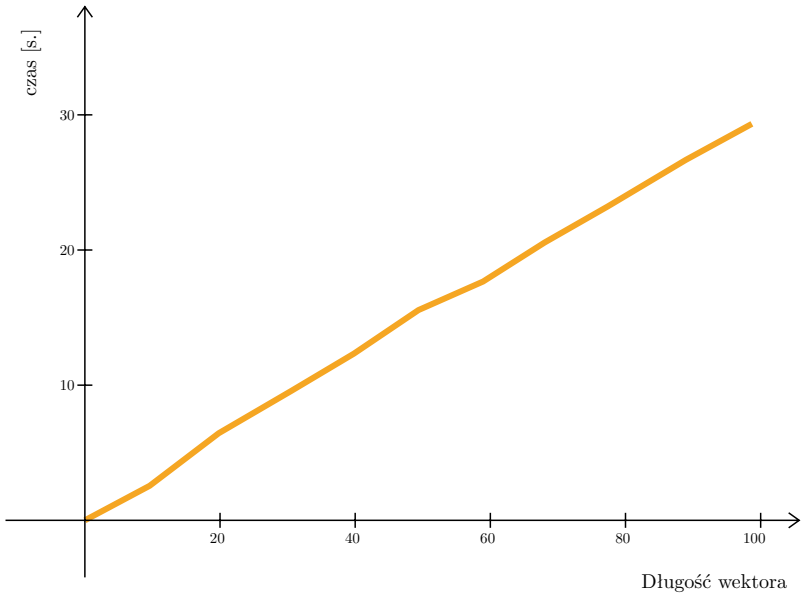
$$Gen(k) = ((0)^*|(1)^*|(2)^*|...|(k)^*)^* \tag{6.1}$$

Funkcja $Gen(k)$ tworzy wyrażenie regularne, dla którego automat NFA po przetłumaczeniu przez funkcję *parseRegex* będzie posiadał ϵ-ścieżki pomiędzy każdą parą stanów. Oznacza to, że analizując kolejne wejścia automatu, algorytm będzie musiał dla każdego z $O(k)$ stanów przeanalizować dodatkowo $k-1$ pozostałych stanów. W tym przypadku osiąga się pesymistyczną złożoność algorytmu: $O(n * k^2)$.

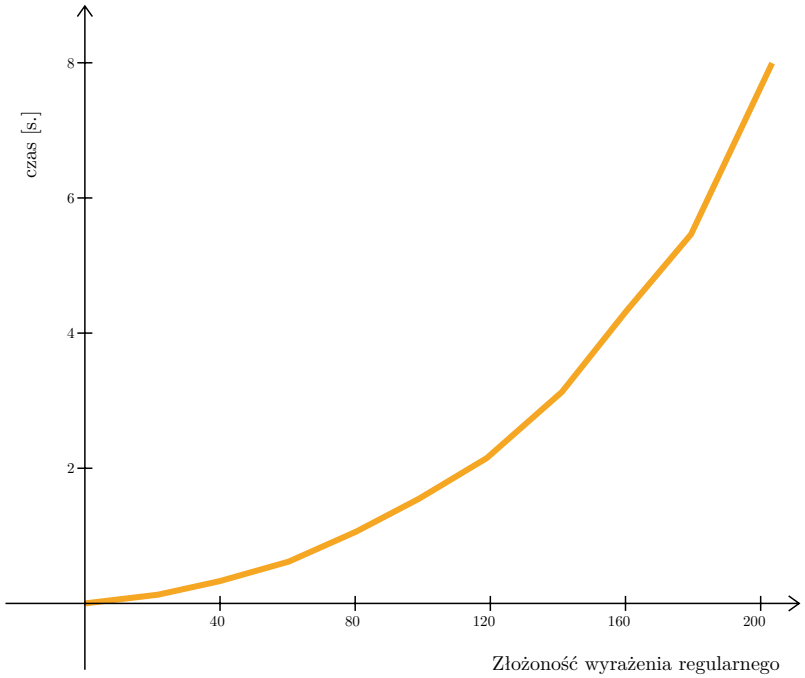
6.2 Rozwiązywanie nonogramów

Jedną z możliwości zastosowania *RegexConstraint* jest rozwiązywanie zagadek logicznych nazywanych *Nonogramami* (tzw. *Griddlers problem*). Polega to na odtworzeniu obrazka na podstawie informacji o liczbie odcinków oraz ich długościach w danym rzędzie/kolumnie. Obrazki mogą być wielokolorowe bądź monochromatyczne.

	(2)	(2)	(1, 4)	(4)	(1)	(4)
(2)			1	1		
(2, 1)	1	1		1		
(4, 1)	1	1	1	1		1
(4)			1	1	1	1
(1, 1)			1			1
(1, 1)			1			1



Rysunek 6.1: Krzywa obrazująca wzrost złożoności czasowej względem długości wejściowego wektora zmiennych. Złożoność wyrażenia regularnego została ustalona na $Gen(700)$.



Rysunek 6.2: Krzywa obrazująca wzrost złożoności czasowej względem rosnącej złożoności automatu wyrażenia regularnego. Długość wektora została ustalona na 300 zmiennych.

Tablica 6.1: Przykład nonogramu i jego rozwiązanie.

Czas rozwiązywania nonogramów jest bardzo zróżnicowany. Jeśli obrazek poddaje się w czasie samej propagacji ograniczeń zanim w ogóle algorytm wyszukiwania zacznie działać to czas działania dla obrazków nawet o boku długości 40 jest mniejszy niż sekunda. Jeśli jednak samo wnioskowanie nie pozwala na ścisłe odnalezienie rozwiązania, tj. po etapie wstępnej propagacji pozostaną jeszcze zmienne o nieustalonym wartościowaniu, to czas działania znacznie się wydłuża ze względu na losowe przydzielanie wartości przez algorytm przeszukujący drzewo rozwiązań.

Rozmiar obrazka	**Czas rozwiązania (sek.)**
10×10	0.01
15×15	0.02
20×20	0.14
25×25	2.81
30×30	16.68

Tablica 6.2: Tabela przedstawiająca wyniki czasowe rozwiązywania losowo wygenerowanych nonogramów.

W załączniku z kodem źródłowym są podane przykłady nonogramów o rozmiarach od 6×6 aż do 50×50. W celu porównania działania algorytmu *RegexConstraint* z istniejącą implementacją predykatu *automaton* w języku *SWI-Prolog* wykonano eksperymenty na wspomnianych nonogramach. Wyniki czasów działania obu algorytmów przedstawia tabela 6.3.

Rozmiar obrazka	**SWI-Prolog**	***RegexConstraint***
6×6	0.022 s.	ok. 0.00 s.
10×10	0.056 s.	0.01 s.
20×20	0.235 s.	0.02 s.
50×50	4.79 s.	0.42 s.

Tablica 6.3: Tabela przedstawia porównanie wyników działania programu opartego o predykat *automaton* napisany w SWI-Prologu oraz czas rozwiązywania modelu napisanego przy użyciu ograniczenia *RegexConstraint*.

Podsumowanie

Możliwość opisu globalnych ograniczeń przy pomocy wyrażeń regularnych wyraźnie zmniejsza skomplikowanie modeli, które dają się zapisać w takiej formie. Dobór odpowiednich algorytmów sprawia też, że tego typu modele można rozwiązać o wiele szybciej niż przy pomocy „konwencjonalnych" metod. Pomimo użycia dość prostych metod wyniki są zaskakująco dobre i to potencjalnie może sprawić, że użycie tego typu ograniczeń stanie się bardziej rozpowszechnione. W celu dalszego zwiększenia wydajności można poszukać lepszych, bardziej szczegółowych rozwiązań dotyczących kompilacji wyrażeń regularnych do automatów skończonych - wiele z takich metod została już wcześniej opisana.

W implementacji języka SWI-Prolog znajduje się predykat o nazwie *automaton*. Jako argumenty przyjmuje opis automatu NFA, wektor liczników oraz tzw. sygnatury, czyli relacje pomiędzy kolejnymi elementami ciągu wejściowego. Działanie tej metody jest oparte o mechanizmy wnioskowania wbudowane w sam język (programowania w logice). W przedstawionej pracy została zaprezentowana metoda napisana w ściśle imperatywnym stylu. Nie ma żadnych przeszkód, aby użytą w tym przypadku gramatykę wyrażeń regularnych wzbogacić o użycie liczników, a sam algorytm filtrujący o ich obsługę. Pozostawia to dalszą możliwość rozwoju tej metody.

Bibliografia

[1] N. Beldiceanu, M. Carlsson, R. Debruyne, T. Petit. Reformulation of global constraints based on constraints checkers. *Constraints*, 10(4):339–362, 2005.

[2] J.-C. Regin. Global constraints and filtering algorithms. M. Milano, redaktor, *Constraint and Integer Programming*, rozdzia/l 4, strony 89–135. Springer, Boston, MA, Oxford, 2004.

[3] F. Rossi, P. van Beek, T. Walsh, redaktorzy. *Handbook of Constraint Programming*. Elsevier Science Inc., New York, NY, USA, 2006.

[4] G. Rozenberg, A. Salomaa, redaktorzy. *Handbook of Formal Languages, Vol. 1: Word, Language, Grammar*. Springer-Verlag, Berlin, Heidelberg, 1997.

[5] G. Rozenberg, A. Salomaa, redaktorzy. *Handbook of Formal Languages, Volume 2. Linear Modeling: Background and Application*. Springer, 1997.

[6] G. Xing. Minimized thompson nfa. *Int. J. Comput. Math.*, 81:1097–1106, 09 2004.

Zawartość płyty CD

Płyta CD zawiera:

- Pracę dyplomową w formie elektronicznej - format PDF.
- Kody źródłowe biblioteki zawierające implementację *RegexConstraint*
- Przykłady użycia ograniczenia *RegexConstraint*

W głównym katalogu zawierającym wszystkie dodatki zawarty jest plik *Makefile*. Do jego uruchomienia potrzebne jest środowisko posiadające polecenie 'make'. W celu skompilowania plików przykładowych należy posłużyć się poleceniem:

```
$ make compile CPX_PATH=[path to IBM CPLEX] \
        SYSTEM=[name of directory with static libs in IBM CPLEX]
```

Następnie aby uruchomić pliki testowe należy użyć polecenia:

```
$ make test
```

Główny katalog zawiera m. in. katalogi *Src* oraz *Inc* zawierające pełną implementację omawianego ograniczenia. Katalog test zawiera pliki testowe, służące do zaprezentowania działania zaproponowanej implementacji. W katalogu *Doc* umieszczona została praca dyplomowa w wersji PDF.