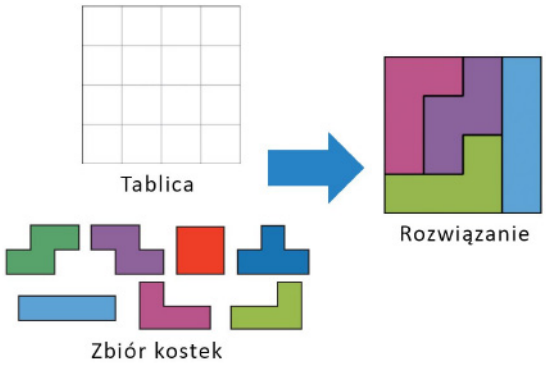
Efektywny algorytm Knutha dla problemu Dokładnego Pokrycia i jego zastosowanie w trudnych łamigłówkach

Niejeden z nas grał, a na pewno zna łamigłówki Sudoku czy Polyomino. To drugie od czasu wprowadzenia przez Solomona Golomba wzbudziło zainteresowanie matematyków zajmujących się nauką i rekreacją. Liczne gry – jak np. Tetris czy Ubongo – łamigłówki i nierozwiązane problemy oparte są na tych zachwycających elementach, które powstają przez połączenie wzdłuż krawędzi wielu nie nachodzących na siebie kwadratów jednostkowych [18].

Tetromino to przypadek szczególny Polyomino, w którym kostki składają się z czterech kwadracików, tak jak we wspomnianych już grach Tetris i Ubongo[1]. Problem polega na tym, żeby z dostępnych elementów wypełnić pewien obszar, np. planszę 4×4 – jak na Rysunku 1. Ponadto zaprojektowanie układu mieszkania czy budynku można sprowadzić do rozwiązania Polyomino, gdzie każdy element odpowiada pokojowi czy pomieszczeniu. Tego typu zagadnienie podjęli autorzy pracy [5].



Rysunek 1. Przykład układanki Tetromino (źródło: [5])

Sudoku to również bardzo znana łamigłówka, która powszechnie występuje w wersji 9×9. Opis problemu oraz dowód, dlaczego ta łamigłówka jest trudnym problemem, można znaleźć w [1].

Możemy sobie wyobrazić bezpośredni sposób rozwiązania powyższych gier. Jednakże pokażemy nieco inne, pośrednie podejście. Mianowicie za pomocą algorytmu dla zupełnie innego (na pierwszy rzut oka) kombinatorycznego problemu rozwiążemy Sudoku oraz Tetromino.

PROBLEM DOKŁADNEGO POKRYCIA

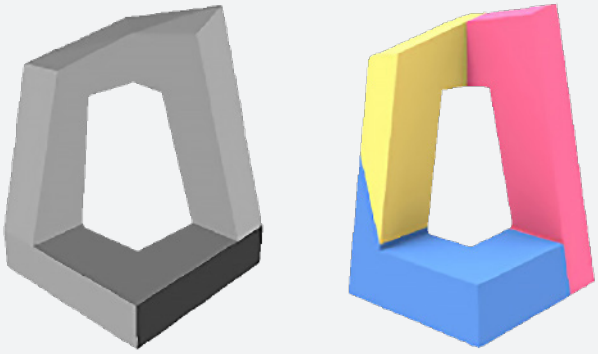
Przedstawimy zatem tak zwany Problem Dokładnego Pokrycia (ECP – Exact Cover Problem), o którym wiemy, że jest obliczeniowo bardzo trudny, należy do znanej klasy problemów NP-zupełnych [19], [22] i nie wiadomo, czy istnieje dla niego algorytm wielomianowy. Opiszemy jednak interesujący algorytm (autorstwa Donalda Knutha) dla tego problemu, który – choć w pesymistycznym przypadku wykładniczy – daje zadowalający czas działania dla niektórych instancji rozważanego problemu.

1. Choć w tej drugiej grze są też dostępne kafelki składające się z dwu, trzech lub pięciu elementów. Odpowiednie ich nazwy to Domino, Tromino i Pentomino.

Puzzle w przemyśle

Na Rysunku 2 widzimy trójwymiarowy obiekt, który zostaje rozłożony na trzy piramidalne (bazowe) części. Niech $P(x, y)$ będzie figurą płaską w przestrzeni, a $f(x, y)$ oznacza pewną funkcję w przestrzeni, której dziedziną jest zbiór P. Kształt jest piramidalny, jeśli jego podstawa jest figurą P, a pozostała część tworzy zbiór punktów należący do funkcji $f(x, y)$ oraz do wysokości poprowadzonej z $f(x, y)$ do punktu podstawy P. Ponadto $f(x, y) - P(x, y) \geq 0$ [2].

Piramidalne kształty są optymalne do formowania, odlewania i warstwowego drukowania 3D. Wiele typowych obiektów nie jest piramidalnych, jak np. wspomniany obiekt widoczny na Rysunku 2, jednakże po rozłożeniu go na bazowe części umożliwia w efekcie oszczędność czasu i materiału podczas druku 3D.



Rysunek 2. Obiekt przypominający wieżę Centralnej Telewizji Chińskiej (CCTV) w Pekinie (źródło: [14]). Kolorowa figura to przykład optymalnej dekompozycji piramidalnej, na której wyróżniono trzy odrębne części gotowe do wydruku 3D

Na Rysunku 3 rozważamy wersję dwuwymiarową tego problemu. W tym przypadku P jest odcinkiem, a funkcja $f(x)$ to np. łamana[3]. Obiekt pierwszy od lewej to podział pewnego kształtu na siedem bloków bazowych. Pozostałe trzy kolorowe obiekty to przykładowe rozwiązania, najlepszy z nich składa się z dwóch bazowych części (niebiesko-zielony). Możemy je również oznaczyć jako zbiory: $\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{1, 2\}, \{1, 3\}, \{3, 4\}, \{4, 6\}, \{5, 6\}, \{6, 7\}, \{1, 2, 3\}, \{3, 4, 6\}, \{4, 5, 6\}, \{4, 6, 7\}, \{5, 6, 7\}, \{3, 4, 6, 7\}, \{4, 5, 6, 7\}$. Zatem najlepsze rozwiązanie składa się z rodziny dwóch zbiorów $\{1, 2, 3\}$ i $\{4, 5, 6, 7\}$ [3].

Rysunek 3. Dekompozycja obiektu na 7 piramidalnych części oraz kolorowe przykłady dekompozycji na mniejszą liczbę elementów bazowych (źródło: [3])

2. Zakładamy, że figura jest po transformacji geometrycznej, w której P leży w płaszczyźnie OXY oraz Z=P(x,y) = 0 i oś OZ jest skierowana do góry, natomiast f(x, y) >= 0.
3. Taka, która spełnia definicję funkcji.

Ponadto za pomocą algorytmu dla ECP można efektywnie rozwiązać inne znane problemy – takie jak piramidalna dekompozycja, projektowanie nowoczesnych układów scalonych czy apartamentowców. Co więcej, zagadnienia z matematyki rekreacyjnej, takie jak wspomniane wcześniej Sudoku czy Polyomino, dzięki algorytmowi dla ECP okażą się w praktyce nie takie trudne.

Wejściem dla problemu ECP jest pewien skończony zbiór U (uniwersum) i pewien zbiór podzbiorów zbioru U – innymi słowy rodzina zbiorów $\mathcal{F}$. Celem jest wybranie takich elementów rodziny $\mathcal{F}$, które są parami rozłączne, a i ich suma daje U. Inaczej mówiąc, wybrana rodzina zbiorów tworzy podział zbioru U.

Rozważmy dla przykładu zbiór U, składający się z m początkowych liczb naturalnych oraz rodzinę zbiorów $\mathcal{F}$ składającą się z n elementów. Niech $m = 7$, wtedy $U = \{1, 2, 3, 4, 5, 6, 7\}$ oraz $\mathcal{F} = \{\{3, 5, 6\}, \{1, 4, 7\}, \{2, 3, 6\}, \{1, 4\}, \{2, 7\}, \{4, 5, 7\}\}$. Oznaczmy też przez $\mathcal{F}_i$, gdzie $1 \leq i \leq n$, kolejne zbiory z $\mathcal{F}$, np. $\mathcal{F}_2 = \{1, 4, 7\}$. Na pewno do rozwiązania nie należą jednocześnie zbiory $\mathcal{F}_1$ oraz $\mathcal{F}_3$, gdyż np. element 3 należy do nich obu.

Zatem ostatecznym rozwiązaniem może być kolekcja zbiorów $\mathcal{F}_1$, $\mathcal{F}_4$, $\mathcal{F}_5$, które rzeczywiście są parami rozłączne oraz tworzą podział zbioru U.

ALGORYTM NIEDETERMINISTYCZNY

Aby znaleźć dokładne pokrycie zbioru U, przedstawimy najpierw niedeterministyczną wersję algorytmu, tj. w każdym jego kroku (dokładnie w wywołaniu rekurencyjnym) zgadujemy[4], jaki zbiór z rodziny zbiorów $\mathcal{F}$ wybieramy. Tak więc algorytm bierze na wejście rodzinę zbiorów $\mathcal{F}$ i wybiera pewien zbiór elementów z $\mathcal{F}$ (o ile istnieje), który spełnia warunki zadania. Po jego zakończeniu weryfikujemy poprawność rozwiązania.

Niedeterministyczny algorytm dla problemu ECP ma następującą postać:

Jeśli zbiór U jest pusty, to wyświetl rozwiązanie R i zakończ algorytm
Jeśli zbiór $\mathcal{F}$ jest pusty, to brak rozwiązania
W przeciwnym wypadku wybierz niedeterministycznie zbiór $\mathcal{F}_i$ (dla pewnego $1 \leq i \leq n$) z rodziny $\mathcal{F}$, taki że $\mathcal{F}_i$ jest podzbiorem U. Jeśli taki zbiór nie istnieje, to nie ma rozwiązania
Usuń zbiór $\mathcal{F}_i$ z rodziny zbiorów $\mathcal{F}$
Dołącz $\mathcal{F}_i$ (lub po prostu numer zbioru) do częściowego rozwiązania R oraz usuń z U wszystkie elementy zbioru $\mathcal{F}_i$
Powtarzaj ten algorytm rekurencyjnie na zredukowanym zbiorze U oraz $\mathcal{F}$[5]

ALGORYTM DETERMINISTYCZNY, CZYLI ALGORYTM BRUTE-FORCE DLA PROBLEMU ECP

Konsekwencją tego, że rozwiązujemy problem NP-trudny, jest fakt, że prawdopodobnie każdy deterministyczny algorytm będzie oparty o proste (siłowe) wyszukiwanie w pewnej przestrzeni rozwiązań, którą tutaj jest rodzina pewnych zbiorów. Przypuśćmy, że mamy już pewne rozwiązanie częściowe. Stoimy zatem przed wyborem pewnego zbioru z $\mathcal{F}$, który spełnia warunki zadania. Takich wyborów może być wiele, więc dla każdego z nich z osobna szukamy rozwiązania. Gdy w pewnym momencie okaże się, że nie możemy dokonać poprawnego wyboru oraz zbiór U nadal nie jest w całości pokryty, musimy się cofnąć do najbliższego miejsca, z którego możemy podjąć inną decyzję. Zauważmy, że ten sposób przejścia po zbiorach przypomina przeszukiwanie drzewa w głąb i występuje w praktyce jako algorytm z nawrotami (ang. *back-tracking*).

Można by zapytać, czy w ogóle warto podejmować próby rozwiązywania problemu, o którym wiemy, że jest NP-trudny i prawdopodobnie najlepszy algorytm będzie działać i tak w czasie wykładniczym, chyba że P = NP. Typowy algorytm wykładniczy ma często złożoność postaci $O(2^n)$ lub $O(n!)$, co oznacza, że np. algorytm o złożoności $O(2^n)$ działa na współczesnym, typowym komputerze szybko dla $n < 27$, ale dla kolejnych n czas wykonania jest już jednak bardzo zauważalny. Nie zawsze jednak tak musi być! Może się okazać, że nasz algorytm działa w czasie $\mathrm{O}(2^{\frac{n}{c}})$ i wówczas takie rozwiązanie może działać szybko dla $n < 27 \times c$, co może być zadowalające przy odpowiednio dużej stałej c.

POSTAĆ MACIERZOWA ECP

Problem Dokładnego Pokrycia można też przekształcić do wygodniejszej postaci macierzy binarnej.

Przypomnijmy zatem, że zbiór $U = \{1, 2, 3, 4, 5, 6, 7\}$ oraz $\mathcal{F} = \{\{3, 5, 6\}, \{1, 4, 7\}, \{2, 3, 6\}, \{1, 4\}, \{2, 7\}, \{4, 5, 7\}\}$.

Kolumny macierzy będą reprezentować U. Natomiast o wierszach macierzy będziemy myśleć jak o kolejnych zbiorach $\mathcal{F}_i$. Na przykład zbiór $\mathcal{F}_2 = \{1, 4, 7\}$ będziemy reprezentować jako $\{1, 0, 0, 1, 0, 0, 1\}$, tj. stawiamy jedynkę wszędzie tam, gdzie odpowiedni numer kolumny reprezentuje element ze zbioru $\mathcal{F}_i$. Kolumny można też oznaczyć kolejnymi literami alfabetu, tj. $U = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}, \mathrm{G}\}$, wtedy np. $\mathcal{F}_2 = \{\mathrm{A}, \mathrm{D}, \mathrm{G}\}$. Spójrzmy na Rysunek 4, na którym mamy pełną reprezentację rodziny $\mathcal{F}$ w postaci macierzy binarnej.

$$\begin{pmatrix} A & B & C & D & E & F & G \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Rysunek 4. Macierz zero-jedynkowa, dla której przykładowym pokryciem są wiersze o numerach (licząc od góry) 1, 4, 5 (źródło: [2])

Ostatecznie więc problem ECP brzmi: czy istnieje zbiór wierszy zawierający dokładnie jedną jedynkę w każdej kolumnie?

Jeśli n jest liczbą wierszy, a m kolumn macierzy, to wtedy liczba wszystkich wyborów wierszy wynosi 2^n. Z góry więc złożoność algorytmu można oszacować przez $O(m2^n)$. Ponieważ w każdym węźle trzeba sprawdzić, czy nie pokryliśmy całego wiersza, stąd dodatkowy czynnik m. Zauważmy jednak, że zwykle nie musimy sprawdzać każdego wyboru.

4. Taki model wyboru w praktyce prawdopodobnie nie istnieje. Odpowiedni algorytm mógłby działać na deterministycznym komputerze z nieograniczoną liczbą równoległych procesorów. Za każdym razem, kiedy możliwy jest więcej niż jeden wybór, tworzone są osobne procesy dla każdego z nich. Gdy jeden z takich potomnych procesów kończy się sukcesem, zwracamy rozwiązanie [17].

5. Przy rozważaniu algorytmów w modelu niedeterministycznym pytamy tylko, czy istnieje taki ciąg zgadywań, który da nam wynik w pożądanym czasie. W codziennym świecie programowania szybkość jego działania zależeć będzie od tego, jak dobrze zgadujemy.

ALGORYTM X KNUTHA

Algorytm podany przez Donalda Knutha nie jest tutaj wyjątkiem – w sensie jego wykładniczej złożoności obliczeniowej – ponieważ także jest oparty o przeszukiwanie z nawrotami[6]. Jednakże jego implementacja jest bardzo efektywna i działa szybko w praktyce.

Aby znaleźć wszystkie dokładne pokrycia, Knuth zaproponował rozwiązanie, które nazwał Algorytmem X. Wejściem jest macierz zero-jedynkowa M, dla której algorytm znajduje pewien podzbiór wierszy (o ile istnieje), który spełnia warunki zadania. W języku zbiorów szukamy odpowiedni podzbiór rodziny zbiorów $\mathcal{F}$, taki że żadne dwa zbiory nie mają elementu wspólnego oraz ich suma daje zbiór U.

Tym razem niedeterministyczny wybór dotyczy wiersza macierzy, co odpowiada równoważnie wyborowi pewnego zbioru $\mathcal{F}_i$ z rodziny $\mathcal{F}$. Możemy zatem albo zgadnąć odpowiedni wiersz, albo utworzyć tyle instancji algorytmu, ile jest możliwości wyboru wiersza – a więc tyle, ile jest jedynek w danej kolumnie nr u – a potem każdy z nich działa niezależnie. Równoważnie wybór takiej kolumny odpowiada pewnemu niewybranemu jeszcze elementowi u z U, a następnie zgadujemy zbiór z rodziny $\mathcal{F}$ spośród wszystkich, do których należy u.

Każdy taki „podalgorytm" dziedziczy macierz M, którą też redukuje ze względu na wybór wiersza r. Jeśli jednak cała kolumna c nie ma jedynek, wtedy proces kończy się niepowodzeniem, co oznacza brak rozwiązania. Z drugiej strony oznacza to, że nie znaleźliśmy żadnego zbioru $\mathcal{F}_i$ z elementem u.

Zwróćmy uwagę na to, że istotną różnicą między tym algorytmem a bezpośrednim brute-force jest to, że wybieramy kolumnę do usunięcia, a nie pojedynczy wiersz. Tzn. zamiast wybierać kolejno, które podzbiory wykorzystamy w naszym rozwiązaniu, skupiamy się na pytaniu: „jak pokryć kolejny element u?". Zauważmy, że w takim podejściu po wybraniu wiersza w tej macierzy możemy w tym wywołaniu rekurencyjnym „nie zajmować" się innymi wierszami, które mają jedynki w tej kolumnie. Szczegóły zostaną przedstawione w dalszej części artykułu dotyczącej algorytmu pokrycia kolumny.

Wybór kolumny c w sposób systematyczny znajdzie rozwiązanie, jednak nie każdy sposób wyboru pozwoli dojść do rozwiązania szybko. Okazuje się, że najlepiej wybierać taką spośród wszystkich, która ma najmniejszą liczbę jedynek (tzn. najmniej zbiorów, które zawierają ten element, tj. u).

Niedeterministyczny algorytm dla problemu Dokładnego Pokrycia ma następującą postać [2]:

Jeśli macierz M jest pusta, to wyświetl rozwiązanie i zakończ algorytm
W przeciwnym wypadku wybierz deterministycznie kolumnę c
Wybierz niedeterministycznie wiersz r, taki że $M[r,\ c] = 1$
Dołącz r do częściowego rozwiązania
Dla każdego j takiego, że $M[r,\ j] = 1$
    Usuń kolumnę j z macierzy M
    Dla każdego i takiego, że $M[i,\ j] = 1$
        Usuń wiersz i z macierzy M
Powtarzaj ten algorytm rekurencyjnie na zredukowanej macierzy M

6. Algorytm z nawrotami jest o tyle lepszy od siłowego sprawdzania wszystkich możliwości, ponieważ w praktyce przestrzeń poszukiwań jest znacznie mniejsza. Choć pesymistycznie też wykładnicza.

Jak zatem efektywnie reprezentować macierz M oraz operacje usuwania i przywracania kolumn i wierszy? Kluczem jest Algorytm DLX (Dancing Links).

DANCING LINKS, ALGORYTM DLX

Knuth zastosował ciekawą oraz bardzo efektywną implementację statycznej[7] czterokierunkowej listy zwanej *dancing links*, która świetnie nadaje się do podejścia *back-tracking* oraz operacji wykonywanych przez Algorytm X.

DANCING LINKS W DZIAŁANIU

W tej części artykułu pokażemy działanie struktury na przykładzie listy dwukierunkowej. Wyróżnijmy pewien element x listy. Niech $x.prev$ oznacza poprzednik elementu x na liście, a $x.next$ następnik elementu x. Wtedy (zdefiniowana dalej) operacja *Usuń(x)* usuwa element x z listy – co, według Knutha, wie każdy programista. Jednak bardzo niewielu programistów wie, że ciąg instrukcji wykonanych jedna po drugiej – w funkcji *Przywróć(x)* – jest przywróceniem elementu x, a więc cofnięciem wcześniejszej operacji wstawienia[8]. Na potrzeby algorytmu *back-tracking* ta niezbyt przydatna na pierwszy rzut oka instrukcja okaże się niezbędna i bardzo efektywna.

//Funkcja usunięcia elementu x z listy
Usuń(x)
    $x.next.prev = x.prev$
    $x.prev.next = x.next$

//Funkcja wstawienia/przywrócenia (przed chwilą usuniętego) elementu x do listy
Przywróć(x)
    $x.next.prev = x$
    $x.prev.next = x$

Idea przywracania elementu x była wprowadzona w [8], gdzie autorzy pokazali, że dzięki temu dobrze znany program Dijkstry dotyczący problemu N-królowych działa prawie dwa razy szybciej.

Przykład listy początkowej i usunięcie z niej elementu C przedstawiono na Rysunku 5. Zauważmy, że powstał „mały" bałagan, tj. C wskazuje na elementy B oraz na D. Zostawmy to jednak, ponieważ ten nieporządek przyda się przy odzyskiwaniu elementu C. Piękno operacji wstawienia polega na tym, że – aby ją wykonać – wystarczy znać wartość elementu x!

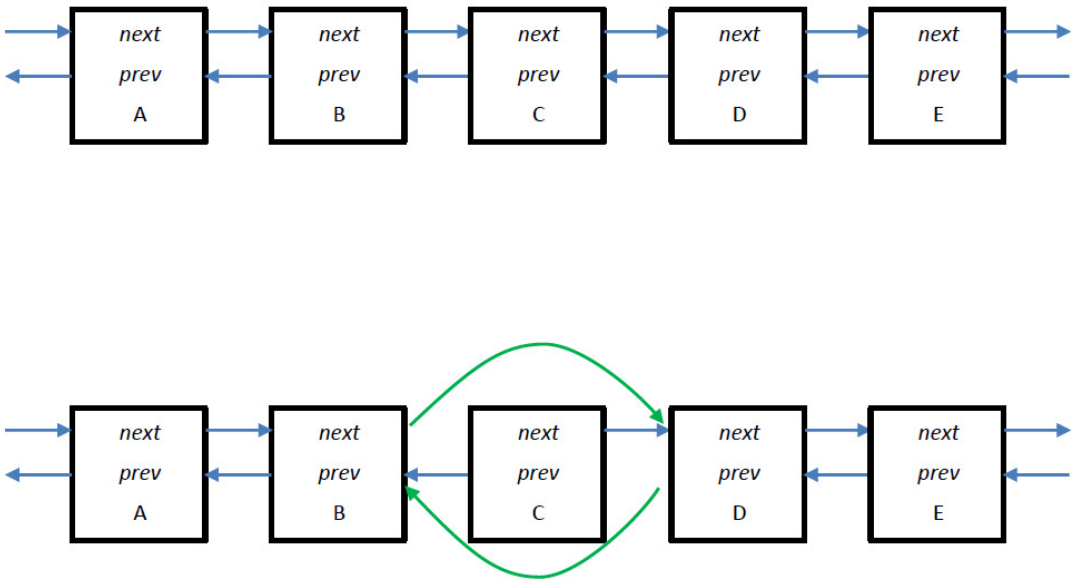
ALGORYTM DLX

Każdy z elementów macierzy wejściowej M reprezentujących jedynkę to węzeł x należący do pewnej listy. Dokładniej należy on do czterokierunkowej listy cyklicznej[9]. Możemy sobie ją wyobrazić jako dwie

7. Generalnie listy, jakie znamy, są dynamiczne. Jednak w tym przypadku struktura utworzona jest tylko raz na początku. W czasie działania algorytmu istotnie wykonujemy operacje „usuń", ale fizycznie nie usuwamy elementów z pamięci, jak np. znane instrukcje free/release/delete.

8. Ważne, żeby przywracać elementy w kolejności „stosowej"/odwrotnej do jego usuwania ze struktury, w przeciwnym wypadku operacja przywracania elementu może zadziałać niepoprawnie.

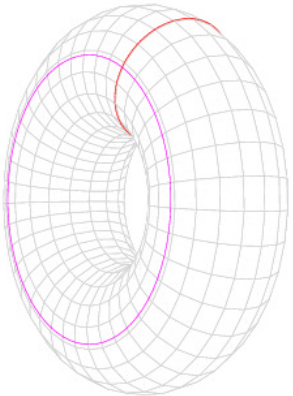
9. Lista nie musi być cykliczna, ale upraszcza to implementację algorytmu.



Rysunek 5. Przykład początkowej cyklicznej listy dwukierunkowej oraz po usunięciu elementu C. Zauważmy, że dwa wskaźniki z C do B oraz z C do D zostały nienaruszone. Są jednak niezbędne do przywrócenia elementu C do listy

osobne listy cykliczne, z polami *prev*/*next* oraz *up*/*down*. Pola *prev* oraz *next* pozwolą przechodzić między kolumnami wzdłuż wiersza M, natomiast pola *up* oraz *down* między wierszami wzdłuż kolumny M. Mamy również dodatkową listę kolumn, w której będziemy pamiętać aktualną liczbę jedynek w kolumnie (pole *cnt* początkowo równe 0). Do elementu tej listy, który jest również nagłówkiem odpowiedniej kolumny, wskazuje pole *columns*, które jest dostępne w każdym węźle reprezentującym jedynkę. Struktura ma również dodatkowy element zwany korzeniem *root*, który wskazuje pierwszą nieusuniętą kolumnę od lewej, tj. pierwszy dotąd nieusunięty węzeł z dodatkowej listy kolumn.

Aby zwizualizować i lepiej zrozumieć tę strukturę danych, dobrze sobie wyobrazić torus[10] (Rysunek 6). Dzięki zastosowaniu tej struktury algorytm staje się bardzo efektywny. W roku 2010 algorytm ten został empirycznie potwierdzony jako najszybsze znane rozwiązanie dla problemu znalezienia wszystkich dokładnych pokryć [4], [15].



Rysunek 6. Torus – przykład poglądowy (źródło: [16])

10. Powierzchnia obrotowa generowana przez obrót okręgu w przestrzeni trójwymiarowej wokół prostej leżącej w płaszczyźnie tego okręgu i nie przecinającej go [16].

Posiadamy już potrzebne struktury danych, żeby zaimplementować cały Algorytm X, zwany dalej DLX. Kluczową operacją jest pokrycie kolumny, tj. usuwamy c z dodatkowej listy kolumn oraz usuwamy wszystkie wiersze, które mają jedynki w kolumnie c. Jednak wcześniej szukamy odpowiedniej kolumny c z najmniejszą liczbą jedynek, przechodząc po dodatkowej liście kolumn.

Usuń z wiersza (x)
$x.down.up = x.up$
$x.up.down = x.down$

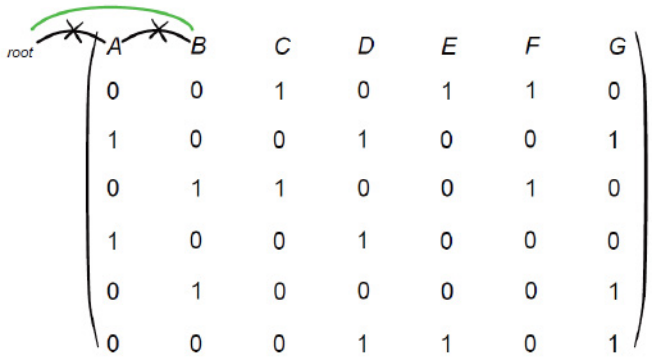
Przywróć do wiersza (x)
$x.down.up = x$
$x.up.down = x$

Opis algorytmu znalezienia optymalnej kolumny c z najmniejszą liczbą jedynek:
//Find c
$c = root.next$
Dla każdego $u \in \{c.next,\ c.next.next,\ ...\} \setminus \{root\}$
if $(u.cnt < c.cnt)\ c = u$

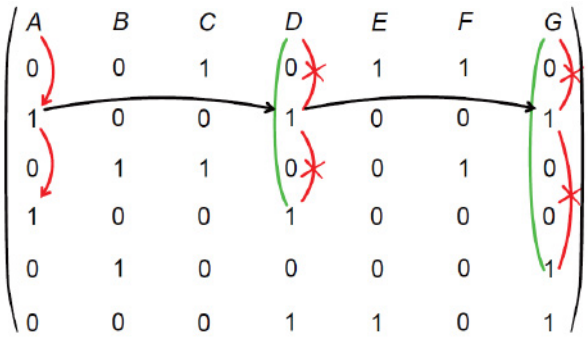
Opis algorytmu usunięcia/pokrycia kolumny jest następujący:
// Usuń/Pokryj kolumnę c
Usuń(c)

// Usuń wszystkie wiersze, które mają jedynkę w kolumnie c
// Nie możemy ich dodać do rozwiązania, bo mielibyśmy
// dwie jedynki w jednej kolumnie.
Dla każdego $u \in \{c.down,\ c.down.down,\ ...\} \setminus \{c\}$
Dla każdego $v \in \{u.next, u.next.next, ...\} \setminus \{u\}$
Usuń z wiersza (v)
$v.columns.cnt{-}{-}$

Przykład poglądowy dla pokrycia pierwszej kolumny A z macierzy z Rysunku 4 znajdziemy na Rysunkach 7a oraz 7b.



Rysunek 7a. Pokrycie kolumny A – faza pierwsza: usunięcie kolumny A z dodatkowej listy (nagłówek macierzy). Odpowiednie dowiązania zostały usunięte, natomiast nowe zaznaczono kolorem zielonym. Dla uproszczenia nie zaznaczamy szczegółowo dowiązań jak na Rysunku 5



Rysunek 7b. Pokrycie kolumny A – faza druga (po pierwszym obrocie zewnętrznej pętli): usunięcie wiersza {A,D,G}. Wykreślamy dowiązania zaznaczone kolorem czerwonym, na zielono natomiast zaznaczono nowe dowiązania. Zauważmy, że elementów z kolumny A nie usuwamy z odpowiadającego wiersza

Proces dokładnie odwrotny, czyli cofnięcie pokrycia kolumny c, ma postać:

Dla każdego $u \in \{c.up,\ c.up.up,\ \ldots\} \setminus \{c\}$
  Dla każdego $v \in \{u.prev,\ u.prev.prev,\ \ldots\} \setminus \{u\}$
    Przywróć do wiersza (v)
    $v.columns.cnt{+}{+}$
Przywróć(c)

Możemy już zapisać ostateczną postać algorytmu wyszukującego wszystkie dokładne pokrycia w postaci pseudokodu poniżej:

Szukaj(k):
  Jeśli nie ma więcej kolumn do pokrycia, to wypisz rozwiązanie i powróć z tego wywołania
  Wybierz kolumnę c z najmniejszą liczbą jedynek
  Jeśli nie ma takiej kolumny, to nie ma rozwiązania i powróć z tego wywołania
  Pokryj kolumnę c
  Dla każdego $r \in \{c.down,\ c.down.down,\ \ldots\} \setminus \{c\}$
    Ustaw $rozwiązanie(k) = r$
    Dla każdego $j \in \{r.next,\ r.next.next,\ \ldots\} \setminus \{r\}$
      Pokryj kolumnę $j.columns$
    Szukaj$(k+1)$
    Usuń r z $rozwiązanie(k)$
    Dla każdego $j \in \{r.prev,\ r.prev.prev,\ \ldots\} \setminus \{r\}$
      Przywróć kolumnę $j.columns$
  Przywróć kolumnę c

Przykład działania algorytmu dla macierzy z Rysunku 4 przedstawiono na Rysunku 9.

PRZYKŁAD ROZWIĄZANIA TETROMINO ZA POMOCĄ ECP

Kiedy mamy już w ręku silne narzędzie, jakim jest algorytm DLX, możemy przystąpić do rozwiązania właściwej łamigłówki. Najpierw pokażemy przykład konwersji układanki Tetromino w wersji 4×5 do ECP. Dodatkowo załóżmy, że mamy nieograniczoną ilość kostek każdego rodzaju. Ponumerujmy – licząc od 1 do rozmiaru układanki – każde pole kolejno wierszami od góry, a następnie kolumnami od lewej do prawej.

Nasze rozwiązanie będzie mieć formę macierzową. Jej kolumny odpowiadają kolejnym polom planszy według numeracji na Rysunku 8. Każdy wiersz tej macierzy odpowiada natomiast konkretnemu ustawieniu kostki na planszy. Jeśli obszar kostki leży na polu numer p, to w p-tej kolumnie macierzy znajduje się 1. W przeciwnym wypadku ustawiamy 0.

Możemy zakodować od 1 do 8 orientacji każdego z 5 kawałków, rozważamy bowiem transformacje geometryczne, takie jak obroty oraz odwracanie.

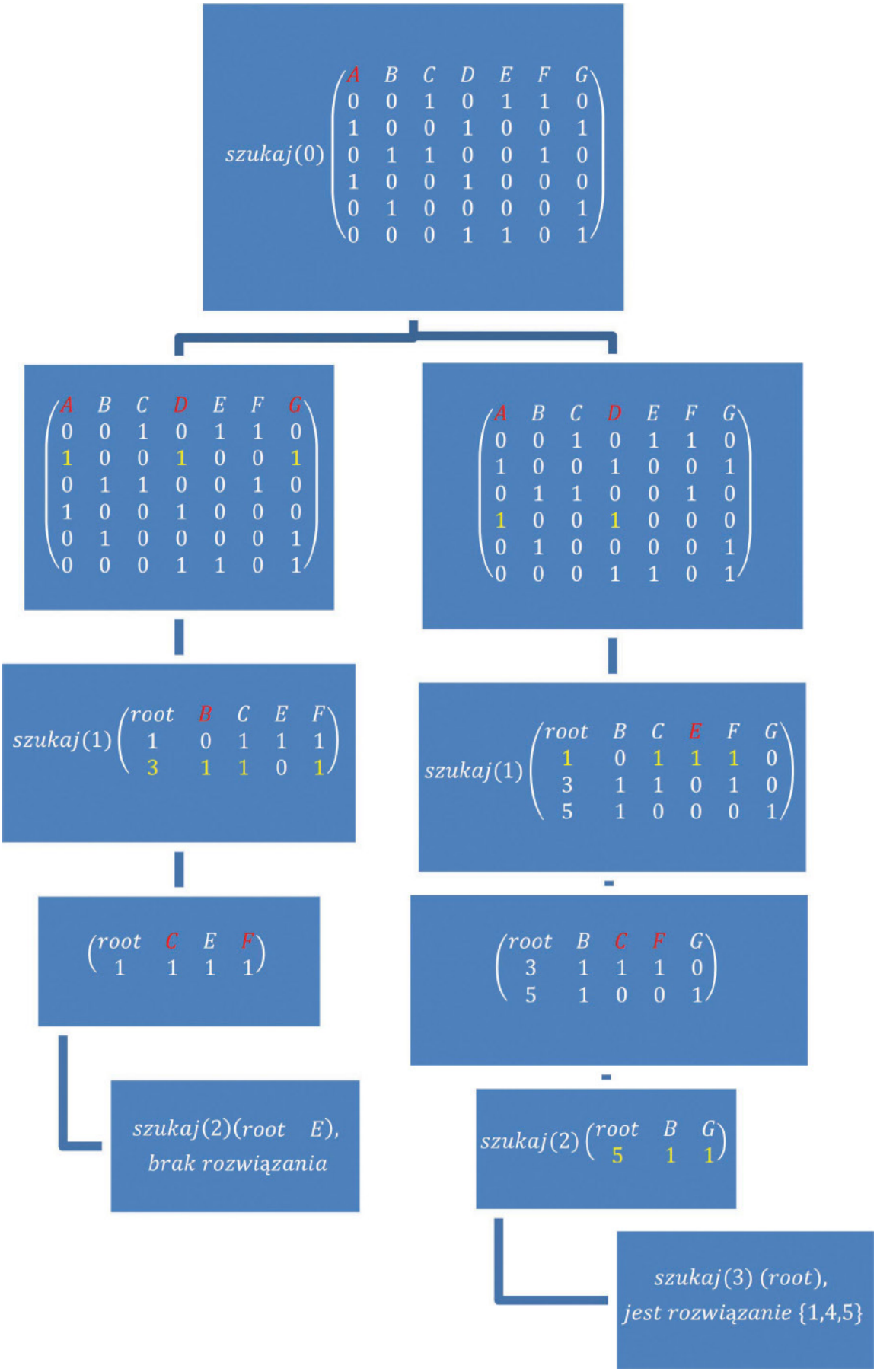
Liczba kolumn M dla układanki z Rysunku 8 jest równa 20. Wierszy M jest tyle, ile wynosi sumaryczna liczba możliwych ustawień każdego z pięciu klocków na układance rozmiaru 4×5 – w tym przypadku 161.

Pytamy teraz, czy istnieje dokładne pokrycie dla M. Jeśli tak, to możemy też rozwiązać Tetromino, w przeciwnym wypadku rozwiązanie nie istnieje. Można też z góry założyć, jakie rodzaje z 5-ciu kawałków puzzli chcemy wykorzystać oraz jakiej maksymalnej ilości[11] przez odpowiednią modyfikację macierzy M, a dokładnie przez dołożenie odpowiednich kolumn.

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

Rysunek 8. Wiersz macierzy M dla zaznaczonych na czerwono kwadratowych fragmentów tworzących kostkę „Z" ma postać: (0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0)

11. Można też dodatkowo zablokować niektóre pola planszy i w rezultacie potrafimy rozwiązać wszystkie łamigłówki w grze Ubongo.



Rysunek 9. Przykład przechodzenia drzewa poszukiwań przez algorytm Szukaj(0) dla macierzy z Rysunku 4. Na żółto zaznaczone są kolejno wybierane wiersze rozwiązania. W drugiej gałęzi przykładowym rozwiązaniem są wiersze o numerach 1, 4, 5. Na czerwono oznaczone są usuwane kolumny

PRZYKŁAD ROZWIĄZANIA SUDOKU ZA POMOCĄ ECP

Dla kolejnej łamigłówki redukcja jest nieco trudniejsza. Konwertujemy Sudoku w wersji 9×9 do *Dokładnego Pokrycia* poprzez utworzenie macierzy rozmiaru 729×324. Dane liczby w Sudoku są reprezentowane przez odpowiednie wiersze macierzy. Matryca koduje wszystkie możliwości wstawienia cyfry do siatki Sudoku (tzn. każdy wiersz reprezentuje decyzję: „wstawiam cyfrę C w polu P"). Ponieważ jest 9 liczb oraz 81 komórek, stąd wszystkich wyborów jest $9 \times 81 = 729$. Macierz ma zatem 729 wierszy. Każdy z tych wierszy zawiera dokładnie 4 jedynki odpowiadające 4 warunkom, które są spełnione, kiedy liczba jest umieszczona w siatce. Pokażemy, jak dokładnie rozmieścić każdą z 4 jedynek w danym wierszu:

1. Zapewniamy dokładnie jedną cyfrę w danej komórce, tj. jedynka w komórce (A, a), tj. wierszu A, kolumnie a lub dwójka w polu (A, a), … lub dziewiątka w polu (A, a), jedynka w polu (A, b) lub dwójka w polu (A, b), … lub dziewiątka w polu (A, b), … jedynka w komórce (I, i), … lub dziewiątka w komórce (I, i). Potrzebujemy do tego zatem 9 × 9 kolumn macierzy. Zauważmy, że dla każdej cyfry w konkretnym polu (X, y) jedynka będzie stać w tej samej kolumnie odpowiadającej komórce (X, y).
2. Zapewniamy, że w każdym wierszu (A-I) planszy Sudoku znajdzie się dokładnie jedno wystąpienie każdej z dziewięciu cyfr. Zatem potrzebujemy kolejno komórki: jedynka w wierszu A, dwójka w wierszu A, trójka w wierszu A … I tak dla każdego wiersza. Potrzebujemy znów 9 × 9 kolumn macierzy.
3. W taki sam sposób radzimy sobie z kolumnami planszy (a-i), dodając kolejne 9 × 9 kolumn macierzy.
4. Podobnie kodujemy kwadraty/regiony 3×3 na planszy (oznaczone cyframi rzymskimi I-IX), dodając kolejne (ostatnie już) 9 × 9 kolumn macierzy.

Daje to nam 324 kolumny macierzy.

Dla zobrazowania, wstawienie siódemki w prawym górnym rogu środkowego regionu – tak jak na Rysunku 10 – odpowiada wierszowi macierzy, w którym:

1. Jedynka jest na polu odpowiadającym cyfrze 7 w polu (D, f) (kolumna 3 × 9 + 6 w pierwszej grupie kolumn).
2. Jedynka jest na polu odpowiadającym cyfrze 7 w wierszu „D" (kolumna 3 × 9 + 7 w drugiej grupie kolumn).
3. Jedynka jest na polu odpowiadającym cyfrze 7 w kolumnie „f" (kolumna 5 × 9 + 7 w trzeciej grupie kolumn).
4. Jedynka jest na polu odpowiadającym cyfrze 7 w regionie „V" (kolumna 4 × 9 + 7 w czwartej grupie kolumn).

TESTOWANIE ALGORYTMU I SZYBKOŚCI DZIAŁANIA

W Tabeli 1 przedstawiono czasy wykonania różnych wersji algorytmu DLX oraz siłowego[12]. Dla układanki Sudoku 9×9 (z pewnym początkowym rozwiązaniem częściowym) algorytm DLX-opt (tj. wybieramy heurystykę kolumny z najmniejszą liczbą jedynek) działa do 3 razy szybciej niż algorytm siłowy.

12. Który też jest oparty o technikę back-tracking.

	a	b	c	d	e	f	g	h	i
A									
B		I			II			III	
C									
D						7			
E		IV			V			VI	
F									
G									
H		VII			VIII			IX	
I									

Rysunek 10. Przykład planszy Sudoku, na której ustawiamy cyfrę 7 w wierszu D, kolumnie f oraz regionie V

	BF	BF-opt	DLX	DLX-opt	#rozwiązań
Tetromino 4×5	≤ 0,001	-	≤ 0,001	≤ 0,001	454
Tetromino 6×6	0,48	-	0,37	0,26	178939
Tetromino 8×5	1,1	-	1,6	1,1	800290
Tetromino 5×8	3	-	3,5	1,1	800290
Tetromino 8×6	44,93	-	57,54	32,51	22483347
Tetromino 6×8	85,44	-	83,83	32,67	22483347
Tetromino 8×8	-	-	-	-	~19 × 10^9
Sudoku (test1)	45,54	28,67	15,26	13,17	4682736
Sudoku (test2)	117,63	74,62	42,61	25,97	10208980
Sudoku (test3)	-	332,87	270,88	111,3	45448179

Tabela 1. Czas wykonania (w sekundach) programu komputerowego na różnych danych testowych. Plansze w testach dla Sudoku są już częściowo wypełnione. Zwróćmy również uwagę na pomiary czasowe dla plansz Tetromino, w których plansza jest tego samego rozmiaru, ale jest obrócona o 90 stopni, np. 8x5 i 5x8. Dla najmniejszej planszy wynik to średnia arytmetyczna z 10 wywołań programu

Tak więc dzięki Donaldowi Knuthowi typowe Sudoku jest proste nawet dla komputera!

MONTE CARLO W POMIARZE SZYBKOŚCI

Czas działania algorytmu *back-tracking* można mierzyć, nie tylko odmierzając systemowy czas od początku do końca działania programu, ale również zliczając odwiedzone wierzchołki drzewa poszukiwań. Drugi model umożliwia predykcję czasu wykonania algorytmu na podstawie tylko pewnej części tego drzewa, bowiem często czas działania takiego algorytmu może zająć niepraktyczną – jeśli nie astronomiczną – ilość czasu i na pewno nie warto tego bezpośrednio sprawdzać.

Knuth użył zatem metody *Monte Carlo* [11]. Idea polega na losowym przejściu drzewa poszukiwań – to znaczy tak jak w pierwszej wersji algorytmu DLX dokonywaliśmy niedeterministycznych wyborów, tak teraz dokonujemy takiego wyboru w sposób losowy. Opracowana przez Knutha funkcja kosztu daje estymatę na podstawie jednej gałęzi drzewa. Jednak taka estymata może być czasem daleka od prawdziwej i jednym z rozwiązań jest powtórzenie testu wiele razy. Na podstawie problemu skoczka szachowego oraz 1000 obserwacji autor podaje, że estymata liczby węzłów drzewa poszukiwań wynosi w przybliżeniu $31{,}23 \times 10^8$ [13], podczas gdy prawdziwa wynosi ok. $31{,}37 \times 10^8$ [14], a błąd względny pomiaru jest mniejszy niż 0,5%!

CZY MOŻNA JESZCZE SZYBCIEJ...?

Pole do optymalizacji można zauważyć np. dla problemu rozwiązywania Sudoku: tutaj wszystkich możliwych plansz jest ok. $6{,}67 \times 10^{21}$ [15]. To bardzo dużo, ale jeśli skupimy się na *istotnie* różnych[16] planszach, ich liczba jest znacząco mniejsza i wynosi ok. $5{,}47 \times 10^9$ [17]. Widać więc, że z pewnością da się przyspieszyć przeglądanie wszystkich plansz w Sudoku, jeśli tylko umiejętnie będziemy pomijać symetryczne duplikaty. Na przykład ustalmy pierwszy wiersz planszy Sudoku, w którym liczby tworzą ciąg rosnący. Zauważmy, że dla każdego rozwiązania możemy zmieniać dowolnie kolejność kolumn 1-3, 4-6 oraz 7-9 oraz dodatkowo można też zamieniać dowolnie 3 bloki kolumn po 3. Mamy łącznie $3!^4$ takich możliwości i o taki współczynnik można zmniejszyć rozmiar poszukiwań. Autorzy [20] podają lepszy współczynnik i dzięki temu możemy przyspieszyć listowanie wszystkich rozwiązań, ustalając pierwszy region w lewym górnym rogu planszy. Ponieważ takich możliwości jest $9!$, więc wystarczy przejrzeć $\frac{N}{9!}$ plansz, gdzie N jest liczbą wszystkich rozwiązań. W pracy [21] wyliczono dokładną liczbę istotnie różnych rozwiązań. Dzięki temu wystarczy przejrzeć „tylko" ok. $5{,}47 \times 10^9$ możliwych ustawień cyfr w Sudoku, żeby znaleźć liczbę wszystkich możliwości wypełnienia planszy.

A czy możemy przyspieszyć rozwiązanie ogólnego problemu, tj. ECP? I na to odpowiedź jest również pozytywna.

13. A dokładnie 3 123 375 511,1 [11].
14. A dokładnie 3 137 317 290 [11].
15. A dokładnie 6 670 903 752 021 072 936 960.
16. Dwie plansze możemy potraktować jako różne, jeśli można przenumerować symbole na planszy lub zastosować operację obrotu lub odwracania planszy. Jako istotnie różnych jest jeszcze mniej i należy rozważyć dodatkowo: permutacje bloków kolumn/wierszy 1-3, 4-6, 7-9, permutacje kolumn/wierszy 1-3, 4-6, 7-9 [21]. W kombinatoryce jest to zliczanie grup symetrycznych permutacji.
17. A dokładne 5 472 730 538.

ALGORYTM DLX + ZDD

W 2017 roku na konferencji na temat sztucznej inteligencji (AAAI-17) autorzy [4] zaproponowali przyspieszenie metody DLX, zwane algorytmem DXZ, tj. DLX z ZDD. Ich metoda konstruuje Binarny Diagram Decyzyjny z tłumieniem Zera (ZDD), który reprezentuje zbiór rozwiązań podczas wyszukiwania w głąb w algorytmie DLX. ZDD umożliwia efektywne wykorzystanie pamięci podręcznej[18] (tzw. spamiętywanie) w celu przyspieszenia wyszukiwania. Na podstawie różnych eksperymentów proponowana metoda jest o kilka rzędów wielkości szybsza niż DLX – nawet do 10000 razy. W Tabeli 1 nie podaliśmy czasu wykonania rozwiązania Tetromino 8×8, ponieważ czas wykonania jest zbyt długi. Jednak algorytm DXZ wykonuje się na komputerze autorów w 42.62 sekundy!

Algorytm ten jest również pierwszym zdolnym do znalezienia w rozsądnym czasie wszystkich rozwiązań Tetromino dla planszy rozmiaru 12×12! Autorzy potwierdzili dokładną liczbę rozwiązań, która jest rzędu 13×10^{24} [19]! [5].

PODSTAWOWY ALGORYTM ROZPROSZONY

Kiedy już coraz trudniej przyspieszyć algorytm, warto sprawdzić, czy można niektóre jego części zrównoleglić. Możemy zatem rozproszyć obliczenia i na każdym komputerze (węźle) policzyć część rozwiązania. Następnie jeden z wyróżnionych komputerów (master node) po zebraniu tych wyników może ustalić rozwiązanie końcowe.

Najprostszym sposobem, by zrównoleglić algorytm DLX, jest rozpoczęcie przeszukiwania drzewa rozwiązań wszerz (Breadth First Search) do pewnej głębokości, a następnie przesłanie każdego wierzchołka u z tej głębokości na inną maszynę lub procesor. Zauważmy, że gałąź od korzenia do wierzchołka u odpowiada pewnemu częściowemu rozwiązaniu. Należy wziąć to pod uwagę przed uruchomieniem właściwego algorytmu DLX na danym komputerze i również przesłać ten fragment rozwiązania do odpowiedniego węzła [12][20].

18. Aby przyspieszyć algorytm DLX, można zauważyć, że niektóre podproblemy w czasie przeszukiwania drzewa w głąb są takie same. Bezpośrednim podejściem jest memoizacja (spamiętywanie) takich częściowych rozwiązań – kosztem jest natomiast duży narzut pamięciowy. Jednak ZDD efektywnie zmniejsza to zapotrzebowanie, używając tylko stałego rozmiaru pamięci do zapamiętania rozwiązania podproblemu.
19. A dokładnie 13 664 822 582 333 502 156 627 512.
20. To jest najprostsze podejście. Bardziej zaawansowane techniki zrównoleglenia algorytmu, np. tzw. splitting partitions, można znaleźć w [9].

PODSUMOWANIE

W tym artykule pokazaliśmy różne zastosowania algorytmu DLX. Niezwykłą korzyścią jest to, że implementując rozwiązanie dla Sudoku czy Tetromino, nie musimy zmieniać implementacji algorytmu, a tylko dostarczyć odpowiednią macierz, a następnie przekształcić wynik na właściwy format. W świecie algorytmiki i złożoności obliczeniowej taki zabieg nazywamy „redukcją" między problemami (Sudoku do ECP, Tetromino do ECP). W bezpośrednim podejściu jednak trzeba dla każdej łamigłówki zmienić nieco podstawową implementację.

DLX oraz problem ECP znajdują zastosowanie nie tylko w łamigłówkach, ale również na przykład w konstrukcji obwodów elektrycznych (MPL).

MPL (Multiple Patterning Litography) to technika, która przezwycięża ograniczenia litograficzne w procesie wytwarzania układów scalonych. W tej technice układ jest rozkładany na wiele tzw. masek (kolorowych kostek, jak w Polyomino). Ponieważ dotąd modelowano problem za pomocą kolorowania grafów, MPL jest na tyle złożony, że właśnie problem ECP znajduje tu swoje zastosowanie [6]. MPL umożliwia producentom projektować układy poniżej 20 nm [10].

ECP znalazł również zastosowanie w opisanej na początku artykułu dekompozycji trójwymiarowych kształtów. Znalezienie rozwiązania dla odpowiedniej instancji problemu ECP pozwala na oszczędność i efektywność druku 3D. Zanim jednak będzie można przystąpić do rozwiązania problemu ECP, należy wcześniej przeprowadzić odpowiednią dekompozycję kształtu, o czym więcej w [3].

Ponadto przedstawiliśmy różne warianty przyspieszenia podstawowego algorytmu Knutha, prezentując rozproszone podejście do obliczeń, jak i diagram decyzyjny ZDD.

ZDD znalazł na przykład zastosowanie w projektowaniu układów mieszkań, sprowadzając ten problem do układania Polyomino [5], [7]. Dzięki zaaplikowaniu tej struktury algorytm DLX działa nawet do 10000 razy szybciej, niż jego poprzednia wersja. Ponadto algorytm DXZ jest w stanie efektywnie zapisać wszystkie rozwiązania. Co więcej, autorzy nadal pracują nad poprawieniem wydajności struktury danych, tak żeby znajdować optymalne rozwiązania w praktycznych problemach.

Co ciekawe, wszystkich pokryć szachownicy rozmiaru 8×8 kostkami Polyomino z liczbą pól mniejszą niż 4 (tj. Monomino, Domino oraz Tromino) jest ok. 92×10^{21} i dokładna liczba[21] może zostać policzona niemal natychmiast przez utworzenie ZDD rozmiaru raptem 512 227 wierzchołków [13].

Widzimy zatem, że układanka Polyomino służy nie tylko łamaniu głowy, ale też ułatwia modelowanie i projektowanie praktycznych obiektów.

Na koniec warto wspomnieć, że za pomocą ECP można również rozwiązać problem N-królowych czy skoczka szachowego.

Podziękowania dla redaktora Delty, Tomasza Kazany, za dyskusje i rady, które pomogły uprościć rozumowanie w niniejszym artykule.

21. 92 109 458 286 284 989 468 604.

Bibliografia

[1] Łukasz Grządko „Dlaczego niektóre łamigłówki są tak trudne?", Delta, Miesięcznik Uniwersytetu Warszawskiego
http://www.deltami.edu.pl/temat/informatyka/2017/08/24/Dlaczego_niektore_lamiglowki/.
Wersja PDF: http://www.deltami.edu.pl/temat/informatyka/2017/08/25/2017-09-delta-sudoku.pdf
[2] Donald E. Knuth „Dancing Links", Stanford University
[3] Ruizhen Hu, Honghua Li, Hao Zhang, Daniel Cohen-Or „Approximate Pyramidal Shape Decomposition". Publikacja: SIGGRAPH 2014, Asia
[4] Masaaki Nishino, Norihito Yasuda, Shin-ichi Minato, Masaaki Nagata „Dancing with decision diagrams: a combined approach to exact cover". Publikacja: AAAI'17: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence
[5] Masaaki Nishino, Norihito Yasuda, Shin-ichi Minato, Masaaki Nagata „Efficient Algorithm for Enumerating All Solutions to an Exact Cover Problem"
[6] Iris Hui-Ru Jiang, Hua-Yu Chang „Multiple Patterning Layout Decomposition Considering Complex Coloring Rules and Density Balancing" Publikacja: IEEE transactions on computer-aided design of integrated circuits and systems, 2017
[7] Atsushi Takizawa, Yushi Miyata, Naoki Katoh „Enumeration of Floor Plans Based on a Zero-Suppressed Binary Decision Diagram"
[8] Hirosi Hitotumatu, Kohei Noshita „A technique for implementing backtrack algorithms and its application", 1979
[9] Sándor Szabó „Speeding up exact cover algorithms by preprocessing and parallel computation"
[10] https://semiengineering.com/knowledge_centers/manufacturing/patterning/multipatterning/
[11] Donald E. Knuth „Estimating the Efficiency of Backtrack Programs. Mathematics of Computation", Publikacja: Mathematics of Computation, 1975
[12] Jan Magne Tjensvold „Generic Distributed Exact Cover Solver"
[13] Donald E. Knuth „The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part1"
[14] Ilke Demir, Daniel G. Aliaga, Bedrich Benes „Near-convex decomposition and layering for efficient 3D printing"
[15] T. Junttila, P. Kaski „Exact cover via satisfiability: An empirical study", 2010
[16] https://pl.wikipedia.org/wiki/Torus_(matematyka)
[17] https://xlinux.nist.gov/dads/HTML/nondetermAlgo.html
[18] M. S. Klamkin, A. Liu „Polyominoes on the Infinite Checkerboard"
[19] Christos H. Papadimitriou „Złożoność obliczeniowa", tytuł oryginalny: „Computational Complexity"
[20] Bertram Felgenhauer, Frazer Jarvis „Enumerating possible Sudoku grids", 2005
[21] Ed Russell, Frazer Jarvis „There are 5472730538 essentially different Sudoku grids ... and the Sudoku symmetry group", http://www.afjarvis.staff.shef.ac.uk/sudoku/sudgroup.html
[22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein „Wprowadzenie do algorytmów", tytuł oryginalny: „Introduction to Algorithms"

ŁUKASZ GRZĄDKO

Autor przygodę z komputerem zaczął w wieku 8 lat. Mając 10 lat, programował interaktywne gry video w języku Basic. Uczestnik wielu konkursów matematyczno-programistycznych. Swoje umiejętności matematyczne i informatyczne poszerzył i wzmocnił, kończąc studia na Uniwersytecie Wrocławskim. Zainteresowany teoretycznymi podstawami informatyki i matematyką, algorytmami i strukturami danych oraz inżynierią oprogramowania. Ponadto aktywny uczestnik TopCoder Algorithms od ponad 16 lat. Obecnie pracuje jako Software Development Engineer w technologiach LTE i 5G Mobile Broadband we wrocławskim oddziale Nokii.