Zmodyfikowana technika programowania dynamicznego

Lech Madeyski[1], Zygmunt Mazur[2]
Politechnika Wrocławska, Wydział Informatyki i Zarządzania, Wydziałowy Zakład Informatyki
Wybrzeże Wyspiańskiego 27, 50-370 Wrocław

Streszczenie. W artykule przedstawiono metodę (technikę) częściowego spamiętywania *partial memoization* na tle innych technik programowania dynamicznego. Wykorzystuje ona ideę ograniczania ilości przechowywanych rozwiązań. Jest to szczególnie przydatne w przypadkach, w których liczba (rozmiar) podproblemów uniemożliwia zastosowanie standardowych technik programowania dynamicznego. Modyfikacja w stosunku do techniki spamiętywania *memoization* polega na tym, że (podobnie jak w przypadku koncepcji wykorzystania pamięci podręcznej *cache*) pamiętany jest tylko pewien podzbiór rozwiązań. Technikę częściowego spamiętywania z powodzeniem wykorzystano do stworzenia nowego algorytmu na bazie standardowego algorytmu faktoryzacji do obliczania niezawodności *K*-terminali. Wydaje się, że zaprezentowana technika może znaleźć praktyczne zastosowanie przy projektowaniu również innych algorytmów.

Słowa kluczowe: projektowanie algorytmów, programowanie dynamiczne, częściowe spamiętywanie.

1. Technika częściowego spamiętywania

Techniki czy metody projektowania algorytmów są klasyczną dziedziną informatyki, szeroko omawianą w pracach [1, 4-7, 9, 12]. Ich użycie prowadzi często do uzyskania szybkich algorytmów. Jedną z powszechnie stosowanych technik jest technika programowania dynamicznego, która pozwala uniknąć wielokrotnego rozwiązywania tych samych podproblemów. Tworzona jest tablica, w której zapamiętywane są rozwiązania najmniejszych podproblemów (czasami określanych mianem podproblemów elementarnych), a następnie obliczonych na ich podstawie rozwiązań podproblemów większych rozmiarów. Proces ten jest kontynuowany, aż do uzyskania rozwiązania problemu pierwotnego *P*, przy czym raz znalezione rozwiązanie podproblemu zostaje zapamiętane i może być wykorzystywane bez konieczności ponownego obliczania.

Przykładem efektywnego zastosowania techniki programowania dynamicznego może być problem wyznaczania elementów ciągu Fibonacciego. Definicja ciągu jest następująca:

[1] E-mail: madeyski@ci.pwr.wroc.pl.

[2] E-mail: mazur@ci.pwr.wroc.pl.

F(1)=1,

F(2)=1,

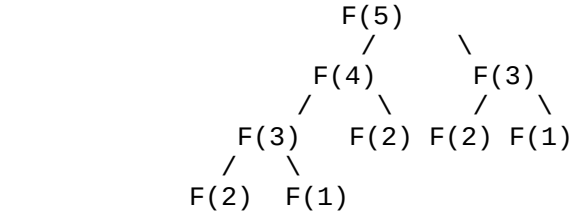
F(n)=F(n-1)+F(n-2).

Formuła ta w naturalny sposób jest wyrażona poprzez następujący algorytm rekurencyjny.

Algorytm 1:

```
int fib(int n)
{
     if(n <= 2)
     return 1;
     else
      return fib(n-1) + fib(n-2);
}
```

Próbując dokładnie prześledzić wywołania rekurencyjne okaże się, że te same podproblemy rozwiązywane są wiele razy:



Złożoność czasowa powyższego algorytmu jest wykładnicza. Stosując technikę programowania dynamicznego można stworzyć prosty i znacznie bardziej efektywny algorytm, który nie będzie wielokrotnie rozwiązywał tych samych podproblemów [12].

Algorytm 2:

```
int fib(int n)
{
   int f[n+1];
   f[1] = f[2] = 1;
   for (int i = 3; i <= n; i++)
     f[i] = f[i-1] + f[i-2];
    return f[n];
}
```

Zarówno złożoność pamięciowa, jak i czasowa tego algorytmu, wynosi $O(n)$. Ponieważ generowanie kolejnych liczb Fibonacciego wcale nie wymaga pamiętania wszystkich wcześniej rozwiązanych podproblemów, można prosto zredukować złożoność pamięciową powyższego algorytmu do $O(1)$[3].

Algorytm 3:

```
int fib(int n)
{
   int a=1, b=1, c=1;
   for (int i=3; i<=n; i++)
   {
     c=a+b;
```

[3] Stosując trik z potęgowaniem macierzy można uzyskać jeszcze efektywniejszy algorytm o złożoności czasowej $O(\log n)$.

```
      a=b;
      b=c;
   }
   return c;
}
```

Czasami zdarza się, że ze względu na naturę rozpatrywanego problemu, trudno jest określić na wstępie podproblemy elementarne i ich rozwiązania, a następnie sposób, formułę otrzymania na ich podstawie rozwiązania problemu pierwotnego. W takiej sytuacji można zastosować technikę spamiętywania określaną w języku angielskim terminem *memoization*[4]. Jest to zmodyfikowana technika programowania dynamicznego wykorzystująca strategię zstępującą. Algorytm wykorzystujący technikę spamiętywania ma formę algorytmu rekurencyjnego z dodanymi funkcjami zapamiętywania identyfikatorów podproblemów wraz z ich rozwiązaniami[5] oraz przeszukiwania zarejestrowanych podproblemów, co zapobiega wielokrotnemu ich rozwiązywaniu.

Algorytm wykorzystujący iteracyjną technikę programowania dynamicznego ze strategią wstępującą, charakteryzuje się podobną złożonością czasową jak algorytm stosujący technikę spamiętywania (różnica o stały czynnik, na korzyść programowania dynamicznego, ze względu na „koszt" wywołań rekurencyjnych i powrotów).

Gdy rozwiązanie problemu wymaga rozwiązania jedynie niektórych spośród wszystkich możliwych podproblemów, to zastosowanie strategii zstępującej zamiast wstępującej zapewnia, że rozpatrywane będą tylko te podproblemy, których rozwiązanie jest niezbędne. Dzięki temu algorytm charakteryzować się może mniejszą, niż w przypadku zastosowania strategii wstępującej, złożonością pamięciową i czasową.

W niektórych przypadkach liczba (rozmiar) podproblemów uniemożliwia zastosowanie standardowych technik programowania dynamicznego. Dlatego zaproponowano technikę częściowego spamiętywania *partial memoization* wykorzystującą ideę ograniczania ilości przechowywanych rozwiązań[6]. Techniki spamiętywania jak i częściowego spamiętywania są odmianami techniki programowania dynamicznego, wykorzystującymi strategię zstępującą. Technika częściowego spamiętywania umożliwia ograniczenie złożoności pamięciowej algorytmu. Modyfikacja w stosunku do techniki spamiętywania polega na tym, że nie są przechowywane wszystkie rozwiązania, a jedynie te, które – jak przypuszczamy – mogą

---

[4] Por. [5], s.358.

[5] Jeżeli jest określona relacja pomiędzy pozycjami tabeli a podproblemami, to wystarczy zapamiętywać jedynie rozwiązania.

[6] Można sobie wyobrazić również iteracyjne programowanie dynamiczne wykorzystujące ideę ograniczania ilości przechowywanych rozwiązań.

znacząco wpłynąć na przyspieszenie algorytmu (np. ostatnio zarejestrowane rozwiązania – jeżeli przypuszczamy, że będą one częściej wykorzystywane niż te zarejestrowane wcześniej). Koncepcja ta jest podobna do koncepcji wykorzystania pamięci podręcznej *cache*[7].

W przypadku zastosowania techniki częściowego spamiętywania należy pamiętać, że:

- strategia ograniczania ilości przechowywanych rozwiązań zależy od charakteru rozwiązywanego problemu;
- w przypadku, gdy istnieją różne strategie dekompozycji problemu pierwotnego o efektywności algorytmu może decydować wybrana strategia dekompozycji.

Algorytmy wykorzystujące technikę rekurencyjną, w których te same podproblemy są rozwiązywane wielokrotnie, można bardzo prosto zmodyfikować, tak, aby wykorzystywały techniki spamiętywania lub częściowego spamiętywania w zależności od charakteru problemu pierwotnego *P* i dostępnej pamięci. Pseudo-kod algorytmu wykorzystującego proponowaną technikę częściowego spamiętywania jest następujący:

```
P(N) – rozwiązywany problem
solve(P(N))
{
   if(P(N)jest zarejestrowany lub jego rozmiar jest wystarczająco mały)
   return rozwiązanie;
   else
   {
      „Podziel” P(N)na mniejsze podproblemy:  P(N1), ... , P(Nk)
      for(i=1; i<=k; i++)
      {
         oblicz i stosując strategię ograniczania ilości przechowywanych
         rozwiązań odrzuć bądź zarejestruj (ewentualnie kosztem wcześniej
         zarejestrowanego rozwiązania) rozwiązanie cząstkowe wi=solve(P(Ni))
      }
      return „Połącz” rozwiązania(w1, ..., wk);
   }
}
```

Problem pierwotny *P* podlega dekompozycji na *k* podproblemów mniejszych rozmiarów, z których każdy podlega dekompozycji tak długo, aż rozmiar problemu stanie się tak mały, że rozwiązanie będzie oczywiste lub będzie można wykorzystać już zarejestrowane rozwiązanie. Raz znalezione rozwiązanie pewnego podproblemu może zostać zarejestrowane i w miarę możliwości później wykorzystywane, jeżeli w trakcie pracy algorytmu okaże się, że

---

[7] Słowo *cache* nie było dotąd używane w kontekście technik programowania. Pamięć *cache* występuje na różnych poziomach architektury komputerów. Są w niej przechowywane często używane dane, co znacznie przyspiesza dokonywane operacje. Pamięć *cache* pierwszego poziomu jest najczęściej umiejscowiona w procesorze. Pamięć *cache* drugiego poziomu, o większej pojemności i nieco wolniejsza, znajduje się zwykle poza procesorem. Często stosuje się tzw. *cache* dysku, który może być realizowany sprzętowo (kontrolery dysków lub same dyski zawierają pamięć cache) i programowo (np. smartdrv z systemu operacyjnego MS-DOS). W przypadku opisywanej techniki w pamięci *cache* będą przechowywane rozwiązania podproblemów i ewentualnie (jeżeli to będzie konieczne) odpowiadające im identyfikatory.

w poszczególnych węzłach drzewa obliczeń występują te same podproblemy. Rozwiązania podproblemów są łączone w celu uzyskania rozwiązania problemu pierwotnego *P*.

Podsumowując, stosowanie techniki częściowego spamiętywania można polecić, gdy:

- mamy do czynienia z wielokrotnym rozwiązywaniem tych samych podproblemów;
- trudno jest określić na wstępie podproblemy mniejszych rozmiarów, ich rozwiązania, a następnie formułę otrzymania na ich podstawie rozwiązania problemu pierwotnego *P*;
- liczba (rozmiar) możliwych podproblemów jest bardzo duża (np. wykładnicza przestrzeń podproblemów) ze względu na mniejszą złożoność pamięciową techniki częściowego spamiętywania niż klasycznego programowania dynamicznego czy techniki spamiętywania.

Ponadto jeżeli rozwiązanie problemu wymaga rozwiązania jedynie niektórych spośród wszystkich możliwych podproblemów, to zastosowanie techniki spamiętywania lub częściowego spamiętywania zapewnia, że rozpatrywane będą tylko te podproblemy, których rozwiązanie jest niezbędne.

2. Przykład efektywnego wykorzystania techniki częściowego spamiętywania

Technikę częściowego spamiętywania wykorzystano do stworzenia nowego i znacznie efektywniejszego algorytmu na bazie standardowego algorytmu faktoryzacji do obliczania niezawodności *K*-terminali. Analiza niezawodności różnego rodzaju sieci, których łącza ulegają wzajemnie niezależnie losowym uszkodzeniom[8] jest tematem intensywnych badań. Najczęściej rozważany problem niezawodności *K*-terminali (*K*-terminal network reliability problem) określa prawdopodobieństwo, iż wszystkie węzły w pewnym określonym zbiorze *K* (2<=|*K*|<=|*V*|) są połączone poprzez ścieżki nie uszkodzonych łączy [8, 11]. Równie często są rozważane przypadki brzegowe:

- problem niezawodności dwóch terminali (2-terminal network reliability), gdy |*K*|=2;
- problem niezawodności wszystkich terminali (All-terminal network reliability), gdy |*K*|=|*V*|.

Zwykle stosowanym algorytmem do wyznaczania niezawodności sieci jest algorytm faktoryzacji z zachowującymi niezawodność redukcjami grafu [8, 11]. Algorytm faktoryzacji wykorzystuje zdarzenia elementarne sprawności lub niesprawności pojedynczej krawędzi.

[8] Na ogół zakłada się, że węzły sieci są całkowicie niezawodne.

Stany grafu można podzielić na dwa zbiory ze względu na dwa możliwe stany krawędzi e_i o niezawodności p_i. Stąd niezawodność *K*-terminali można wyrazić w postaci prostej formuły niezawodności warunkowej:

$$R(G_K) = p_i R(G_K | e_i \text{ funkcjonuje}) + (1 - p_i) R(G_K | e_i \text{ nie funkcjonuje}).$$

Twierdzenie faktoryzacji jest topologiczną interpretacją powyższej formuły dla grafów nieskierowanych.

Twierdzenie 1 (Twierdzenie faktoryzacji)

Zgodnie z formułą faktoryzacji niezawodność *K*-terminali sieci probabilistycznej reprezentowanej poprzez graf *G* z wyróżnionym podzbiorem węzłów *K* można wyrazić następująco:

$$R(G_K) = p_i R(G_{K'} * e_i) + (1 - p_i) R(G_K - e_i),$$

gdzie:

e_i – dowolna krawędź grafu G_K;

p_i – prawdopodobieństwo, że łącze reprezentowane przez $e_i \in E$ funkcjonuje;

$G_{K'} * e_i = (V - u - v + w, E - e_i), \quad w = u \cup v;$

$$K' = \begin{cases} K & \text{jeżeli } u, v \notin K; \\ K - u - v + w & \text{jeżeli } u \in K \text{ lub } v \in K; \end{cases}$$

$G_K - e_i = (V, E - e_i).$

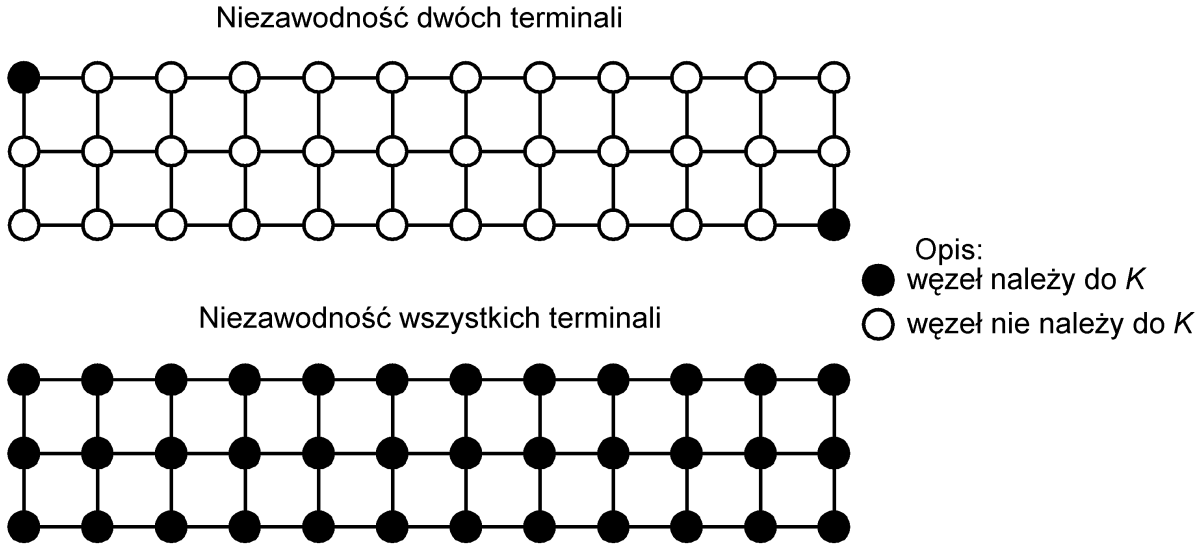
Niezawodność $R(G_K)$ dowolnej sieci reprezentowanej przez graf G_K może być obliczona poprzez rekurencyjne wywoływanie formuły faktoryzacji i ewentualne dokonywanie zachowujących niezawodność redukcji sieci [11]. Pesymistyczna złożoność czasowa takiego algorytmu jest wykładnicza. Ponieważ wykazano [2, 3, 11], że problem niezawodności *K*-terminali w ogólnym przypadku należy do klasy problemów NP-trudnych i #P-zupełnych, więc mało prawdopodobne jest istnienie wielomianowego (w najgorszym przypadku) algorytmu do rozwiązania tego problemu. Możliwe jest jednak zastosowanie w algorytmie faktoryzacji techniki częściowego spamiętywania. W efekcie prowadzi to do znacznego przyspieszenia obliczania niezawodności sieci.

Badając algorytm faktoryzacji można zauważyć, iż w wielu węzłach binarnego drzewa obliczeń te same podproblemy (sieci powstałe w wyniku stosowania faktoryzacji i redukcji) rozwiązywane są wiele razy. Zamiast wielokrotnie rozwiązywać ten sam podproblem (obliczać niezawodność tej samej sieci) jak ma to miejsce w przypadku standardowego

algorytmu faktoryzacji znacznie efektywniej byłoby użyć wcześniej obliczony i zapamiętany rezultat obliczeń.

Załóżmy, iż *Fact* oznacza standardowy algorytm faktoryzacji (wg Page i Perry [8])[9] z zachowującymi niezawodność redukcjami grafu (równoległą, szeregową, pierwszego i drugiego stopnia). Natomiast niech *Fact&Cache* oznacza zaproponowany przez nas algorytm faktoryzacji z zachowującymi niezawodność redukcjami grafu wykorzystujący technikę częściowego spamiętywania. Algorytm *Fact&Cache* używa strategii przechowywania ostatnio zarejestrowanych rozwiązań. Ponadto podproblemy o większym rozmiarze są dłużej przechowywane niż podproblemy o mniejszym rozmiarze. Zarówno algorytm *Fact* jak i *Fact&Cache* zostały praktycznie zaimplementowane w języku C++.

Doświadczalnie porównano efekty pracy programów wykorzystujących algorytmy *Fact*, *Fact&Cache* jak i rezultaty otrzymane za pomocą gotowego pakietu *NRA97* powstałego na uniwersytecie Mittweida[10]. Wykonano obliczenia niezawodności dwóch terminali jak i niezawodności wszystkich terminali w przypadku sieci 3x12 z poniższego rysunku.



Rysunek 1. Sieć 3x12.

Program wykorzystujący algorytm *Fact&Cache* okazał się ponad sto razy szybszy od pakietu NRA97, jak również programu wykorzystującego algorytm *Fact* w przypadku obliczania niezawodności wszystkich terminali i przynajmniej kilka tysięcy razy szybszy w przypadku obliczania niezawodności dwóch terminali. Pozwala to przypuszczać, że

---

[9] Strategia wyboru krawędzi do faktoryzacji została zmieniona z losowej na stałą (zawsze wybierana jest ostatnia krawędź z listy krawędzi grafu), co daje na ogół lepsze rezultaty.

[10] NRA97 (Network Reliability Analysis (C)1997, HTW Mittweida) – pakiet do analizy niezawodnościowej sieci, pracujący w środowisku Windows 3.1/Windows95 stworzony na Uniwersytecie Mittweida

zaprezentowana technika częściowego spamiętywania może znaleźć praktyczne zastosowanie w projektowaniu algorytmów do rozwiązywania wielu innych problemów. Szczególnie w przypadkach, w których liczba (rozmiar) podproblemów ogranicza zastosowanie technik spamiętywania czy programowania dynamicznego.

Literatura:

[1] A. V. Aho, J. E. Hopcroft, J. D. Ullman, *Projektowanie i analiza algorytmów komputerowych*, PWN, 1983.

[2] M. O. Ball, J. S. Provan, The complexity of counting cuts and of computing the probability that a graph is connected, *SIAM J. Comp.*, 1983 (12), 777-788.

[3] M. O. Ball, Computational complexity of network reliability analysis: An overview, *IEEE Trans. Reliability*, 1986 (R-35), 230-239.

[4] L. Banachowski, K. Diks, W. Rytter, *Algorytmy i struktury danych*, WNT, 1996.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Wprowadzenie do algorytmów*, WNT, 1997.

[6] A. Drozdek, D. L. Simon, *Struktury danych w języku C*, WNT, 1996.

[7] E. Horowitz, S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, 1978.

[8] L. B. Page, J. E. Perry, A practical implementation of the factoring theorem for network reliability, *IEEE Trans. Reliability*, 1988 (37) Aug, 259-267.

[9] R. Sedgewick, *Algorithms in C*, Addison-Wesley, 1990.

[10] L. G. Valiant, The complexity of enumeration and reliability problems, *SIAM J. Computing*, 1979 (8), 410-421.

[11] R. K. Wood, Factoring Algotithms for Computing K-Terminal Network Reliability, *IEEE Trans. Reliability*, 1986 (R-35), 269-278.

[12] P. Wróblewski, *Algorytmy, struktury danych i techniki programowania*, Helion, 1997.