A Toolbox for Online Algorithms

(Notatki do minikursu)

Marcin Bieńkowski

phd open, 1-2 kwietnia 2011

1 Funkcje potencjału

Mamy daną listę jednokierunkową ze wskaźnikiem na początek tej listy. Załóżmy, że lista ma w danej chwili długość ℓ. Definiujemy na liście następujące operacje:

- Insert(x): wstawienie elementu x na koniec listy; koszt operacji to $\ell + 1$;
- Access(x): dostęp do elementu x; jeśli x jest na miejscu i-tym, to koszt jest równy i;
- Delete(x): skasowanie elementu x; koszt jak w przypadku operacji Access(x).

> Reorganizacja listy. Wejście dla problemu składa się z sekwencji operacji Insert, Access i Delete. Bezpośrednio po operacji Insert(x) lub Access(x) można za darmo przesunąć x na dowolne miejsce bliżej początku listy. Następnie można dokonać płatnych zamian: zamiana dwóch sąsiednich elementów kosztuje 1. Należy zminimalizować sumaryczny koszt wszystkich operacji.

Algorytm Mtf (Move-To-Front) po każdej operacji Access(x) czy Insert(x) przenosi x (za darmo) na początek listy.

Twierdzenie 1.1 ([ST85])**.** *Algorytm* Mtf *jest* 2*-konkurencyjny.*

Dowód. Ustalmy dowolną sekwencję wejściową σ o długości m. Będziemy śledzić działanie algorytmu optymalnego Opt i algorytmu Mtf na tej sekwencji i porównywać ich koszty. W tym celu zdefiniujemy następującą funkcję potencjału.

Inwersja jest to para elementów x i y, taka, ze x pojawia się przed y w liście algorytmu Mtf, ale po y w liście algorytmu Opt. Potencjał w kroku t, oznaczany przez $\Phi(t)$ definiujemy jako liczbę inwersji w kroku t. Zauważmy, że Φ jest zawsze nieujemne, a jeśli Mtf i Opt zaczynają od takich samych list (np. od pustych) to $\Phi_0 = 0$. Udowodnimy, że w dowolnym kroku $1 \leqslant t \leqslant m$ zachodzi

$$\textsc{Mtf}(t) + \Phi(t) - \Phi(t-1) \leqslant 2 \cdot \textsc{Opt}(t) - 1 \tag{1.1}$$

Jeśli zsumujemy powyższą nierówność po wszystkich krokach t otrzymamy $\textsc{Mtf}(\sigma)+\Phi(m)-\Phi(0) \leqslant 2\cdot\textsc{Opt}(\sigma)-m$, czyli $\textsc{Mtf}(\sigma) \leqslant 2 \cdot \textsc{Opt}(\sigma)$.

W pozostałej części dowodu pokażemy nierówność (1.1). Zauważmy, że dowolny krok t możemy podzielić na dwie akcje:

1. Algorytmy Mtf i Opt płacą za żądanie występujące w kroku t. Następnie Mtf dokonuje ewentualnej (darmowej) zamiany i Opt dokonuje swojej (ewentualnej) darmowej zamiany.
2. Opt dokonuje swoich płatnych zamian.

Dla każdej akcji z osobna udowodnimy, ze (1.1) zachodzi.

Dowód dla drugiej akcji jest łatwy. Niech P oznacza liczbę płatnych zamian wykonywanych przez algorytm Opt, tj. $\textsc{Opt} = P$. Każda taka zamiana sąsiednich elementów powoduje wzrost potencjału o co najwyżej 1. Zatem całkowity wzrost potencjału $\Delta\Phi$ związany z tą akcją jest ograniczony z góry przez koszt Opt.

Dla pierwszej akcji, rozpatrujemy trzy przypadki, ze względu na typ żądania występującego w kroku t. Niech ℓ będzie liczbą elementów listy przed żądaniem w kroku t.

1. Żądanie Insert(x_t). Rzeczywisty koszt dowolnego algorytmu jest identyczny i równy

   $$\textsc{Mtf}(t) = \textsc{Opt}(t) = \ell + 1 \ .$$

   Dodatkowo tworzone jest co najwyżej $\Delta\Phi(t) \leqslant \ell$ nowych inwersji. Zatem (1.1) jest spełniona.

2. Żądanie Access(x_t). Niech A będzie liczbą elementów, które poprzedzają x_t w liście Mtf i liście Opt. Niech B będzie liczbą elementów, które poprzedzają x_t w liście Mtf, ale występują po x_t w liście Opt. Wtedy mamy

   $$\textsc{Mtf}(t) \ = \ A + B + 1 \ , \qquad \textsc{Opt}(t) \ \geqslant \ A + 1 \ .$$

   Jaki jest koszt i zmiana potencjału związana z reorganizacją listy? Zgodnie z założeniem algorytmy Mtf i Opt dokonują darmowych zamian. Przy przesuwaniu x_t na początek listy Mtf usuwamy B inwersji i wprowadzamy co najwyżej A nowych. Zatem zmiana potencjału związana z reorganizacją listy algorytmu Mtf jest ograniczona z góry przez $A - B$. Zatem

   $$\textsc{Mtf}(t) + \Delta\Phi(t) \ \leqslant \ 2A + 1 \ \leqslant \ 2 \cdot \textsc{Opt}(t) - 1$$

3. Żądanie $\mathsf{Delete}(x_t)$. Przyjmijmy identyczne oznaczenia jak w poprzednim podpunkcie. Wtedy również zachodzi $\textsc{Mtf} = A + B + 1$ oraz $\textsc{Opt} \geqslant A + 1$. Po tym żądaniu nie ma możliwości wykonania darmowych zamian. Usunięcie x_t z list powoduje usunięcie B inwersji i nie powoduje utworzenia dodatkowych.

W każdym z trzech powyższych przypadków udowodniliśmy, że (1.1) zachodzi też dla pierwszej akcji. ■

2 Funkcje pracy

Mamy dany graf z odległościami między każdą parą wierzchołków zadanymi przez funkcję $d()$ spełniającą warunek trójkąta. W tym grafie przechowujemy jeden egzemplarz dużego niepodzielnego pliku o rozmiarze $D \geqslant 1$.

> Przenoszenie pliku (*file migration*). Dany jest ciąg wierzchołków v_i, które chcą odwołać się do (fragmentu) pliku w kolejnych krokach. Jeśli w kroku t plik jest w wierzchołku u, to algorytm płaci za to odwołanie $d(v_i, u)$. Następnie algorytm może przenieść plik do dowolnego wierzchołka u', płacąc $D \cdot d(u, u')$. Należy zminimalizować sumaryczny koszt.

W tym rozdziale przedstawimy technikę nazywaną funkcją pracy (*work function*), którą wykorzystamy do konstrukcji optymalnego algorytmu zrandomizowanego dla grafu dwuwierzchołkowego.

Do prezentacji tej techniki, wygodniej jest korzystać z innego opisu algorytmu: opisu rozkładowego. Mianowicie zamiast określać, że algorytm przenosi plik z pewnym prawdopodobieństwem, będziemy definiować algorytm przez jego rozkład prawdopodobieństwa nad możliwymi stanami. W przypadku problemu przenoszenia pliku, w kroku t będziemy określać jaki jest rozkład prawdopodobieństwa pozycji pliku.

Wierzchołki naszego grafu oznaczymy przez a i b. Bez straty ogólności możemy założyć, że odległość między a i b wynosi 1. W każdym kroku będziemy określać rozkład μ; taki rozkład oznacza, że algorytm ma plik w a z prawdopodobieństwem $\mu(a)$ a w b z prawdopodobieństwem $\mu(b) = 1 - \mu(a)$.

Zauważmy, że jeśli algorytm ma w kroku t rozkład μ, a odwołanie jest w wierzchołku a, to w wartości oczekiwanej algorytm płaci za to odwołanie $\mu(a) \cdot 0 + \mu(b) \cdot 1 = \mu(b)$. Musimy teraz pokazać, że przejście między dwoma rozkładami da się wykonać i określić jaki jest koszt takiego przejścia.

Lemat 2.1. *Dla grafu dwuwierzchołkowego (a, b) z $d(a, b) = 1$ i dwóch rozkładów prawdopodobieństwa μ i μ', koszt przejścia pomiędzy nimi wynosi $D \cdot |\mu(a) - \mu'(a)|$.*

Dowód. Załóżmy, że mamy zrandomizowany algorytm, który po przeczytaniu pewnego fragmentu wejścia ma plik w wierzchołku a z prawdopodobieństwem $\mu(a)$ i w wierzchołku b z prawdopodobieństwem $\mu(b)$.

Jeśli $\mu'(a) = \mu(a)$, to algorytm nie przenosi pliku, związany z tym koszt jest równy zero i teza twierdzenia jest spełniona. W przeciwnym przypadku, bez straty ogólności możemy założyć, że $\mu(a) > \mu'(a)$. Wtedy strategia dla algorytmu jest następująca:

- jeśli plik jest w b, nic nie rób,
- jeśli plik jest w a, przenieś go do b z prawdopodobieństwem $p = \frac{\mu(a) - \mu'(a)}{\mu(a)}$.

Zauważmy, że prawdopodobieństwo że algorytm skończy z plikiem w wierzchołku a wynosi $\mu(a) \cdot (1 - p) = \mu'(a)$, a więc otrzymaliśmy zadany rozkład prawdopodobieństwa. Oczekiwany koszt, jaki poniósł nasz algorytm wynosi $p \cdot \mu(a) \cdot D = D \cdot (\mu(a) - \mu'(a))$. ■

Funkcje pracy (*work functions*) są techniką, która często pozwala na konstrukcję algorytmów online osiągających optymalne współczynniki. Idea polega na tym, że w każdym kroku t dla dowolnego stanu x obliczamy $w_t(x)$, koszt optymalnego rozwiązania widzianej do tej pory sekwencji, które kończy działanie w stanie x. Funkcję w_t nazywamy funkcją pracy w kroku t; indeks t będziemy zawyczaj pomijać.

Oczywistą własnością funkcji w jest: $\textsc{Opt}(\sigma) = \min_x w_{|\sigma|}(x)$. Warto zwrócić uwagę, że choć koszt optymalnego rozwiązania na dowolnym prefiksie wejścia o długości ℓ jest równy $\min w_\ell(x)$, to optymalny algorytm nie musi być (a często nie może być) optymalny na każdym z prefiksów. Jednak w analizie algorytmów możemy (i będziemy) zakładać, że koszt $\textsc{Opt}$ w danym kroku t jest równy $\min_x w_t(x) - \min_x w_{t-1}(x)$.

Zależność między funkcją pracy a kosztem optimum prowadzi do pomysłu, że algorytm powinien przebywać możliwie najczęściej w stanie, który minimalizuje funkcję w. Jeśli jednak tak zdefiniujemy algorytm, to zazwyczaj adwersarz jest w stanie wygenerować sekwencję, w której to minimum zmienia się często i algorytm płaci dużo za zmianę stanów.

Przy algorytmach zrandomizowanych można sobie z tym poradzić, zmieniając stan z małym prawdopodobieństwem. Ta idea jest podstawą algorytmu EDGE. Zdefiniujmy przesunięcie $g := w(b) - w(a)$. Zauważmy, że jeśli algorytm optymalny może skończyć z pewnym kosztem $w(a)$ w wierzchołku a, to może skończyć z kosztem co najwyżej $w(a) + D$ w wierzchołku b (może wziąć strategię kończącą w a i na samym końcu przenieść plik do b). Dlatego też $w(b) \leqslant w(a) + D$ i zatem z symetrii wynika, że $g \in [-D, D]$.

Algorytm EDGE ustala pewien rozkład prawdopodobieństwa na podstawie wartości g, mianowicie

$$\mu(a) := \frac{D+g}{2D} \ , \qquad \mu(b) := \frac{D-g}{2D} \ . \tag{2.1}$$

Twierdzenie 2.2 ([CLRW97])**.** *Algorytm* EDGE *jest* $(2 + \frac{1}{2D})$*-konkurencyjny na grafach dwuwierzchołkowych.*

Dowód. Dowód jest tak naprawdę klasyczną analizą zamortyzowaną, choć nie napiszemy bezpośrednio funkcji potencjału. Po pierwsze obliczymy jakie są koszty EDGE i OPT w zależności od zmiany g. Po pierwsze zauważmy, że jeśli g się zmienia, to zmienia się o 1 i wtedy związany z tym oczekiwany koszt przenosin pliku to $D \cdot \frac{1}{2D} = \frac{1}{2}$.

Bez straty ogólności, możemy założyć, że g jest nieujemne, czyli $w(b) \geqslant w(a)$. Rozpatrzmy parę przypadków:

1. Jeśli odwołanie jest w a i $g < D$, to g zwiększa się o 1. W efekcie EDGE płaci $\mu(b)$ za żądanie i 1/2 za przenosiny pliku, zaś OPT nie płaci nic.
2. Jeśli odwołanie jest w a i $g = D$, to g nie zmienia się. Zauważmy, że w takim przypadku $\mu(a) = 1$ i zatem EDGE nie płaci nic. OPT również nic nie płaci.
3. Jeśli odwołanie jest w b i $g = 0$, to jest tak samo jak gdyby odwołanie było w a (patrz przypadek 1).
4. Jeśli odwołanie jest w b i $g > 0$, to g zmniejsza się o 1. W efekcie EDGE płaci $\mu(a)$ za żądanie i 1/2 za przenosiny pliku, zaś OPT płaci 1.

Zobaczmy teraz jak wyglądają koszty w zależności od zmiany $|g|$:

1. Jeśli $|g|$ się nie zmienia, to nic się nie dzieje.
2. Jeśli $|g|$ rośnie o 1, to $\mathbf{E}[\text{EDGE}] = 1 - \frac{g}{2D}$, a $\text{OPT} = 0$.
3. Jeśli $|g|$ maleje o 1, to $\mathbf{E}[\text{EDGE}] = 1 + \frac{g}{2D}$, a $\text{OPT} = 1$.

Zauważmy, że trudny jest przypadek 2, bo wtedy $\text{OPT} = 0$. Zróbmy zatem następujący eksperyment myślowy: W momencie kiedy $|g|$ się zmniejsza (np. z $x+1$ do x) oprócz płacenia za faktyczny oczekiwany koszt EDGE odłożmy dodatkowo kwotę, która zapewni nam, że będziemy mogli zapłacić za zwiększenie $|g|$ od x do $x+1$. Oznacza to, że zamortyzowany koszt w momencie zmniejszenia się $|g|$ to $(1 - \frac{x+1}{2D}) + (1 + \frac{x}{2D}) = 2 + \frac{1}{2D}$, natomiast zamortyzowany koszt w momencie zwiększenia się $|g|$ to 0. Zatem konkurencyjność wynosi $2 + \frac{1}{2D}$.

Ponieważ zaczynamy od maksymalnej możliwej wartości $|g|$, tj. od D, na początku nie musimy mieć odłożonej żadnej kwoty. ■

3 Programowanie liniowe

ONLINE SET COVER [AAA03]. Dane jest skończone uniwersum $\mathcal{U}$. Dana jest również m-elementowa rodzina $\mathcal{F}$ podzbiorów $\mathcal{U}$ pokrywająca całe $\mathcal{U}$. Dla każdego ze zbiorów $S \in \mathcal{F}$, $c_S \geqslant 1$ oznacza jego koszt. Wejście składa się z ciągu elementów $e_1, e_2, e_3, \ldots$. Po każdym elemencie musimy wypisać podzbiór $\mathcal{F}$ pokrywający wszystkie widziane dotychczas elementy. Generowany ciąg podzbiorów $\mathcal{F}$ musi być wstępujący, tj. możemy tylko dodawać zbiory do rozwiązania, a nie wyrzucać.

W tej części przedstawimy jak rozwiązać ułamkowy wariant tego problemu. Oznacza to, że nasze rozwiązanie w każdym kroku jest funkcją $x : \mathcal{F} \to [0, 1]$ (będziemy pisać x_S zamiast $x(S)$) i wymagamy, żeby dla każdego już widzianego elementu e_i zachodziło $\sum_{S \ni e_i} x_S \geqslant 1$. Oczywiście znowu zakładamy, że generowane przez nas rozwiązanie jest „monotoniczne", tj. ułamkowe części zbiorów możemy tylko dodawać do rozwiązania ale nie usuwać z niego.

Wariant offline tego problemu (gdy mamy z góry dane k elementów $e_1, \ldots, e_k$ do pokrycia) jest rozwiązywalny w czasie wielomianowym, gdyż możemy napisać odpowiedni program liniowy, oznaczany niżej przez (P_k). Nasz algorytm online nie będzie go jednak rozwiązywać, lecz posługiwać się pośrednio jego nierównościami.

$$\begin{aligned} \text{zminimalizuj} \quad & \sum_{S \in \mathcal{F}} c_S \cdot x_S & \\ \text{z zachowaniem ograniczeń:} \quad & \sum_{S \ni e_i} x_S \geqslant 1 & \forall_{1 \leqslant i \leqslant k} \\ & x_S \geqslant 0 & \forall_{S \in \mathcal{F}} \end{aligned}$$

Zauważmy, że dobrym rozwiązaniem programu (P_0) jest wzięcie $x_S = 0$ dla każdego S. Takie rozwiązanie ma oczywiście koszt 0. Naszym celem jest generowanie monotonicznych rozwiązań kolejnych programów liniowych $(P_1), (P_2), (P_3), \ldots$, tak żeby ich wartości były odpowiednio małe.

W tym celu napiszemy dualny program liniowy (D_k), tworząc dla i-tego równania $\sum_{S \ni e_i} x_S \geqslant 1$ zmienną y_i. W przypadku tego problemu program dualny będzie służyć nam tylko do analizy, ale często jest on też elementem algorytmu.

$$\begin{aligned} \text{zmaksymalizuj} \quad & \sum_{1 \leqslant i \leqslant k} y_i & \\ \text{z zachowaniem ograniczeń:} \quad & \sum_{e_i \in S} y_i \leqslant c_S & \forall_{S \in \mathcal{F}} \\ & y_i \geqslant 0 & \forall_{1 \leqslant i \leqslant k} \end{aligned}$$

Zauważmy, że wartość optymalnego rozwiązania dla (D_0) jest również równa zeru. Będziemy teraz konstruować ciąg rozwiązań układów (P_i) i (D_i) taki, że prawdziwe są następujące kluczowe warunki.

W1. Rozwiązanie dla (P_i) spełnia wszystkie ograniczenia (P_i).

W2. $\Delta P_i \leqslant 2 \cdot \Delta D_i$, gdzie ΔP_i jest przyrostem kosztu rozwiązania P_i w stosunku do kosztu rozwiązania P_{i-1}. ΔD_i definiujemy analogicznie.

W3. Rozwiązanie dla (D_i) przekracza ograniczenie układu (D_i) co najwyżej o czynnik $O(\log m)$.

Lemat 3.1. *Jeśli algorytm generuje rozwiązania układów (P_i) i (D_i) zgodne z warunkami W1, W2 oraz W3, to jego konkurencyjność wynosi $2 \cdot O(\log m) = O(\log m)$.*

Dowód. Wystarczy pokazać, że dla każdego kroku k wartość rozwiązania układu (P_k) (oznaczmy ją przez P_k^*) jest mniejsza niż $2 \cdot O(\log m) \cdot \textsc{Opt}_k$, gdzie $\textsc{Opt}_k$ jest wartością rozwiązania optymalnego po k-tym kroku. Żądana konkurencyność wynika bezpośrednio ze złożenia dwóch zależności:

$$\begin{aligned} P_k^* &= \sum_{i=1}^{k} \Delta P_i \leqslant \sum_{i=1}^{k} 2 \cdot \Delta D_i = 2 \cdot D_k^* \ , \\ D_k^* &\leqslant \mathcal{O}(\log m) \cdot \textsc{Opt}_k \ , \end{aligned}$$

■

Opiszemy teraz algorytm dla kroku k autorstwa Buchbindera i Naora [BN09b, BN09a] Zauważmy, ze w programie (P_k) pojawił się nowy warunek $\sum_{S \ni e_k} x_S \geqslant 1$ odpowiadający konieczności pokrycia elementu e_k. Natomiast w programie (D_k) pojawiła się nowa zmienna y_k początkowo równa 0 i zmodyfikowane zostały wszystkie warunki $\sum_{e_i \in S} y_i \leqslant c_S$, w których zbiór S zawiera wymagany od teraz element e_k. (Po lewej stronie tych warunków pojawiła się wartość y_k.)

Chcemy teraz zmodyfikować już istniejące rozwiązanie dla (P_{k-1}) i (D_{k-1}). Dopóki $\sum_{S \ni e_k} x_S < 1$ wykonujemy następujące dwie operacje:

1. Dla każdego S takiego, że $e_k \in S$ wykonujemy $x_S \leftarrow (1 + A_S) \cdot x_S + B_S$.
2. $y_k \leftarrow y_k + 1$

Żeby lepiej zrozumieć istniejące zależności, wartości A_S i B_S dobierzemy później. Zauważmy jednak już teraz, że żeby zminimalizować wielkość generowanego rozwiązania układu (D_k) iteracji powinno być mało, a zatem A_S i B_S powinny być duże. Z drugiej strony chcemy porównywać wzrosty rozwiązań (P_k) i (D_k) i zatem chcemy, żeby A_S i B_S były małe.

Twierdzenie 3.2. *Istnieją wartości A_S i B_S dla których powyższy algorytm spełnia warunki W1, W2 oraz W3.*

Dowód. Warunek W1 zachodzi trywialnie. Wykonywana pętla nie psuje już spełnionych warunków, tylko stara się poprawić sytuację $\sum_{S \ni e_k} x_S < 1$. Ponieważ w każdym kroku pętli algorytm zwiększa każdą ze stojących po lewej stronie równania wartości x_S co najmniej o B_S, po skończonej liczbie iteracji otrzymamy $\sum_{S \ni e_k} x_S \geqslant 1$.

Dla warunku W2 zauważmy, że w każdej iteracji pętli wartość rozwiązania układu (D_k) zwiększana jest o 1, natomiast na zwiększenie wartości rozwiązania układu (P_k) wpływa zwiększenie wartości x_S dla zbiorów S, takich że $e_k \in S$. Zatem zmiana w wartości rozwiązania układu (P_k) wynosi

$$\sum_{S \ni e_k} c_S \cdot \Delta x_S = \sum_{S \ni e_k} c_S \cdot (A_S \cdot x_S + B_S) < c_S \cdot (A_S + m \cdot B_S) \ ,$$

gdzie nierówność wynika z tego, że przed aktualizacją zachodziło $\sum_{S \ni e_k} x_S < 1$ oraz z tego, że liczba różnych zbiorów do których należy e_k wynosi co najwyżej m. Wybierając $A_S = 1/c_S$, zaś $B_S = 1/(m \cdot c_S)$ otrzymujemy żądaną zależność $\sum_{S \ni e_k} c_S \cdot \Delta x_S \leqslant 2$.

Pozostaje udowodnić warunek W3. Przyjrzyjmy się dowolnemu warunkowi $\sum_{e_i \in S} y_i \leqslant c_S$ (dla dowolnego $S \in \mathcal{F}$) z programu (D_k). Pokażemy, że jest on przekroczony co najwyżej $O(\log m)$ razy, czyli zachodzi $\sum_{e_i \in S} y_i \leqslant c_S \cdot O(\log m)$. Zauważmy, że w momencie, w którym wzrasta dowolny y_i wchodzący w skład lewej strony tego warunku to algorytm zwiększa też x_S stosując przypisanie

$$x_S \leftarrow \left(1 + \frac{1}{c_S}\right) \cdot x_S + \frac{1}{m \cdot c_S} \ .$$

Wartość x_S przed zwiększeniem była mniejsza od 1, a zatem po zwiększeniu jest mniejsza niż $(1+1) \cdot 1 + 1 = 3$. Oznacza to, że x_S nigdy nie przekroczy wartości 3. Prostą indukcją można pokazać, że po ℓ zwiększeniach wartości x_S zachodzi

$$3 > x_S = \frac{1}{m} \cdot \left[\left(1 + \frac{1}{c_S}\right)^{\ell} - 1\right] \ .$$

A zatem biorąc $\ell = \sum_{e_i \in S} y_i$, otrzymujemy

$$\sum_{e_i \in S} y_i \leqslant \frac{\log(3m+1)}{\log(1 + 1/c_S)} = O(c_S \cdot \log m) \ . \qquad \blacksquare$$

4 Klasyfikuj i wybierz losowo

W tym rozdziale zajmiemy się podproblemem routingu, tj. wyborem tras. Dla uwagi ustalmy pewien graf $G = (V, E)$. Wejście dla problemu routingu składa się z par (s_j, t_j), a dla każdej takiej pary algorytm musi wybrać w grafie G jakąś ścieżkę P_j łączącą s_j z t_j. Interpretacja tego procesu jest taka, że na wejściu pojawiają się żądania nawiązania połączenia telefonicznego z s_j do t_j, a algorytm wybiera trasę, po której ma się to odbywać. Zakładamy, że każda krawędź ma określoną przepustowość.

> DOPUSZCZANIE POŁĄCZEŃ. Każde żądanie (s_j, t_j) algorytm może przyjąć (i wybrać ścieżkę P_j) albo odrzucić. Obciążenie dowolnej krawędzi musi być nie większe od jej przepustowości. Celem jest maksymalizacja liczby przyjętych połączeń.

Ponieważ maksymalne zyski osiągane przez algorytmy są ograniczone, będziemy zajmować się tylko ściśle konkurencyjnymi algorytmami. Co więcej, w tej części rozpatrzymy przypadek prostego grafu: N wierzchołków numerowanych od 0 do $N-1$ połączonych w linię. Wszystkie $N-1$ krawędzi ma przepustowość 1. Taki graf nazywamy N-linią. Wybór ścieżki w takim grafie jest jednoznaczny, więc algorytm decyduje tylko, czy połączenie przyjąć czy odrzucić.

Twierdzenie 4.1. *Dowolny deterministyczny algorytm online dla dopuszczania połączeń na N-linii ma współczynnik konkurencyjności co najmniej $N-1$.*

Dowód. Na początku adwersarz chce połączenia między wierzchołkiem 1 a N. Jeśli algorytm odrzuci to połączenie, to adwersarz kończy sekwencję: optymalny zysk jest równy 1, a zysk algorytmu jest równy 0. Jeśli algorytm przyjmie to połączenie, to adwersarz żąda połączenia między wszystkimi $N-1$ parami sąsiadujących wierzchołków. Wtedy zysk algorytmu to 1, a zysk optymalnego rozwiązania to przyjęcie tych $N-1$ połączeń. $\blacksquare$

Rozważmy zatem następujący zrandomizowany algorytm dla dopuszczania połączeń na N-linii. Dla uproszczenia założymy, że $N = 2^k$. Niech e_1 będzie krawędzią powodującą rozbicie linii na dwie linie równej długości, każdej posiadającej 2^{k-1} wierzchołków i niech $E_1 = \{e_1\}$. Następnie niech $e_{2,1}, e_{2,2}$ są krawędziami które dzielą te dwie linie na połowy równej długości (każda o 2^{k-2} wierzchołkach), a $E_2 = \{e_{2,1}, e_{2,2}\}$. Następnie postępujemy rekurencyjnie otrzymując „dzielące" zbiory $E_1, E_2, E_3, \ldots E_k$. Krawędzie ze zbiorów E_i nazywamy krawędziami poziomu i.

Mówimy, że połączenie należy do poziomu i, jeśli

$$i = \min_j \{E_j \cap E' \neq \emptyset\} \ ,$$

gdzie E' jest zbiorem krawędzi na ścieżce określającej połączenie. Innymi słowy połączenie należy do poziomu i, jeśli nie zawiera krawędzi z poziomu $i-1$, ale zawiera krawędzie z poziomu i.[1]

> Algorytm CRS (Classify-and-Randomly-Select): Wybierz losowo (z jednostajnym rozkładem) poziom $i^* \in \{1, 2, \ldots, \log N\}$. Następnie akceptuj tylko połączenia z poziomu i^* i tylko jeśli nie kolidują z już zaakceptowanymi połączeniami.

Twierdzenie 4.2 ([ABFR94]). *Algorytm* CRS *jest* $\lceil \log N \rceil$*-konkurencyjny.*

Dowód. Dla uproszczenia założymy, że $N = 2^k$ i pokażemy, że algorytm jest $\log N$-konkurencyjny. Zauważmy najpierw, że jeśli wszystkie połączenia w sekwencji wejściowej należałyby do jednego poziomu j, to zachłanne przyjmowanie tych połączeń byłoby optymalną strategią. Żeby to pokazać, zauważmy, że krawędzie poziomu $j-1$ i poziomów wyższych dzielą naszą linię na kawałki, z których każdy zawiera dokładnie jedną krawędź poziomu j. Dowolne połączenie poziomu j zawiera się całkowicie w którymś z tych kawałków i jednocześnie zawiera należącą do tego kawałka jedyną krawędź poziomu j. Zatem dla danego kawałka optymalny algorytm jest w stanie zaakceptować co najwyżej jedno połączenie zawarte w tym kawałku. Jednocześnie algorytm zachłanny też je zaakceptuje.

Teraz niech c_i będzie liczbą połączeń z poziomu i, które są możliwe jednocześnie do zaakceptowania. Wtedy z powyższej obserwacji wynika, że

$$\mathbf{E}[\mathrm{CRS}(\sigma)] = \sum_{i=1}^{\log N} \mathbf{Pr}[\text{CRS wybiera poziom } i] \cdot c_i \ .$$

Niech o_i oznacza liczbę połączeń z poziomu i, które zaakceptował Opt. Oczywiście $o_i \leqslant c_i$ (nierówność pojawia się jeśli połączenia z różnych poziomów kolidują ze sobą). Zatem

$$\mathbf{E}[\mathrm{CRS}(\sigma)] \geqslant \sum_{i=1}^{\log N} \frac{1}{\log N} \cdot o_i = \frac{1}{\log N} \cdot \mathrm{Opt}(\sigma) \ .$$

Można rozszerzyć dowód na dowolne N, niekoniecznie będące potęgą dwójki, otrzymując tezę twierdzenia. ■

Literatura

[AAA03] Noga Alon, Baruch Awerbuch, and Yossi Azar. The online set cover problem. In *Proc. of the 35th ACM Symp. on Theory of Computing (STOC)*, pages 100–105, 2003.

[ABFR94] Baruch Awerbuch, Yair Bartal, Amos Fiat, and Adi Rosen. Competitive non-preemptive call control. In *Proc. of the 5th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 312–320, 1994.

[AGLR94] Baruch Awerbuch, Rainer Gawlick, Frank T. Leighton, and Yuval Rabani. On-line admission control and circuit routing for high performance computing and communication. In *Proc. of the 35th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 412–423, 1994.

[Bar96] Yair Bartal. Distributed paging. In *Dagstul Workshop on On-line Algorithms*, pages 97–117, 1996.

[BN09a] Niv Buchbinder and Joseph Naor. The design of competitive online algorithms via a primal-dual approach. *Foundations and Trends in Theoretical Computer Science*, 3(2–3):93–263, 2009.

[1] W tej definicji zakładamy, że istnieje poziom 0 nie zawierający żadnych krawędzi.

[BN09b] Niv Buchbinder and Joseph (Seffi) Naor. Online primal-dual algorithms for covering and packing. *Math. Oper. Res.*, 34:270–286, 2009. Also appeared in *Proc. of the 13th ESA*, pages 689–701, 2005.

[BS89] David L. Black and Daniel D. Sleator. Competitive algorithms for replication and migration problems. Technical Report CMU-CS-89-201, Department of Computer Science, Carnegie-Mellon University, 1989.

[CLRW97] Marek Chrobak, Lawrence L. Larmore, Nick Reingold, and Jeffery Westbrook. Page migration algorithms using work functions. *Journal of Algorithms*, 24(1):124–157, 1997. Also appeared in *Proc. of the 4th ISAAC*, pages 406–415, 1993.

[Ira91] Sandy Irani. Two results on the list update problem. *Information Processing Letters*, 38(6):301–306, 1991.

[KP95] Elias Koutsoupias and Christos H. Papadimitriou. On the k-server conjecture. *Journal of the ACM*, 42(5):971–983, 1995. Also appeared in *Proc. of the 26th STOC*, pages 507–511, 1994.

[LRWY99] Carsten Lund, Nick Reingold, Jeffery Westbrook, and Dicky C. K. Yan. Competitive on-line algorithms for distributed data management. *SIAM Journal on Computing*, 28(3):1086–1111, 1999. Also appeared as *On-Line Distributed Data Management* in *Proc. of the 2nd ESA*, pages 202–214, 1994.

[ST85] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[Tar85] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.