Algorytmy Online

(Skrypt do wykładu)

Marcin Bieńkowski

Instytut Informatyki Uniwersytetu Wrocławskiego
Wrocław, 2005–2023

Spis treści

1. Wstęp do algorytmów online

W.1A
+10m

Problem optymalizacyjny.
- Zbiór wejść $\mathcal{I}$.
- Dla każdego wejścia $\sigma \in \mathcal{I}$: zbiór *dopuszczalnych* rozwiązań $\mathcal{F}(\sigma)$.
- Funkcja kosztu $\textsc{Cost}(\sigma, s) \in \mathbb{R}_{\geq 0}$ (gdzie $s \in \mathcal{F}(\sigma)$).
- min lub max.

Będziemy rozważać głównie *problemy minimalizacyjne*. Dla algorytmu Alg przez $\textsc{Alg}(\sigma)$ czasem będziemy rozumieć koszt rozwiązania generowanego przez Alg na σ a czasem samo rozwiązanie.

Definicja 1.1. *Rozwiązanie optymalne:*

$$\textsc{Opt}(\sigma) = \min_{s \in \mathcal{F}(\sigma)} \textsc{Cost}(\sigma, s)$$

Definicja 1.2. *Deterministyczny Alg jest R-aproksymacyjny jeśli dla każdego $\sigma \in \mathcal{I}$ zachodzi*

$$\textsc{Alg}(\sigma) \leq R \cdot \textsc{Opt}(\sigma) \ .$$

Algorytm online: algorytm, który dostaje wejście „po kawałku" i „po kawałku" musi wypisywać wyjście.

Definicja 1.3. *Deterministyczny Alg jest* ściśle R-konkurencyjny*, jeśli jest algorytmem online i jest R-aproksymacyjny.*

Uwagi:
- Wciąż porównujemy nasz algorytm Alg z najlepszym możliwym rozwiązaniem *offline* Opt.
- Nie walczymy zazwyczaj z niewystarczającą mocą obliczeniową (jak w aproksymacji) tylko i wyłącznie z nieznajomością przyszłości.

1.1. Problemy typu wypożycz-lub-kup

Problem wypożyczania nart (Ski Rental Problem, SRP). Codziennie rano podejmujemy decyzję, czy narty pożyczyć (za 1) czy też kupić (za $B \in \mathbb{N}$). Jeździmy przez cały dzień a wieczorem łamiemy nogę (i kończymy swoją narciarską karierę) lub też nie. Cel: minimalizacja całkowitego kosztu.

Niech $T \in \mathbb{N}_+ \cup \{\infty\}$ oznacza wybrany przez adwersarza dzień, w którym łamana jest noga

Obserwacja 1.4. *Rozwiązanie optymalne to kupno nart pierwszego dnia jeśli $B < T$, a wypożyczanie nart każdego dnia w przeciwnym przypadku. Zatem $\textsc{Opt}(\sigma) = \min\{B, T\}$.*

Jak konkurencyjna jest strategia „kup pierwszego dnia"? A strategia „zawsze pożyczaj"?

Twierdzenie 1.5. *Niech Alg pożycza narty przez $B-1$ dni, a dnia B je kupi (pod warunkiem, że wcześniej nie złamie nogi). Wtedy Alg jest $(2-1/B)$-konkurencyjny.*

Dowód. Weźmy dowolną sekwencję wejściową σ. Niech T będzie liczbą dni w których narciarz ma jeszcze sprawne nogi (tzn. noga jest łamana pod koniec dnia T).

$$\textsc{Alg}(\sigma) = \begin{cases} T & \text{jeśli } T \leq B-1 \\ B-1+B & \text{jeśli } T \geq B \end{cases} .$$

Porównując to z $\textsc{Opt}(\sigma) = \min\{B, T\}$ otrzymujemy, że dla dowolnego wejścia σ

$$\textsc{Alg}(\sigma) \leq \left(2 - \frac{1}{B}\right) \cdot \textsc{Opt}(\sigma) \ . \qquad \blacksquare$$

Twierdzenie 1.6. *Ścisła konkurencyjność dowolnego algorytmu deterministycznego dla problemu wypożyczania nart wynosi co najmniej $(2 - \frac{1}{B})$.*

Dowód. WLOG, ALG jest zdefiniowany jako „kup narty dnia i", gdzie $i \in \mathbb{N} \cup \{\infty\}$. Co robi adwersarz? Pozwala algorytmowi kupić narty dnia $i < B$ i w dniu i łamie mu nogę. Wtedy:

$$\frac{\text{ALG}(I)}{\text{OPT}(I)} = \frac{i-1+B}{\min\{i,B\}} \geq 2 - \frac{1}{B} \ .$$

■

PROBLEM SPIN-BLOCK. Proces chce uzyskać dostęp do zasobu, który jest aktualnie zajęty. Jądro systemu może najpierw czekać dowolny czas (*spin*) na zwolnienie zasobu, a następnie — jeśli zasób nie zostanie zwolniony — zatrzymać proces (*block*) i przekazać sterowanie do innego procesu, co trwa B ms. Celem jest minimalizacja czasu przeznaczonego na uzyskanie dostępu do zasobu.

Decyzja algorytmu = ile należy czekać? Deterministyczny algorytm online osiągający optymalny współczynnik konkurencyjności 2 jest analogiczny dla SRP: czeka B ms na zwolnienie zasobu. Można go poprawić stosując randomizację.

Definicja 1.7. *Randomizowany algorytm online jest R-konkurencyjny, jeśli dla każdego wejścia σ zachodzi*

$$\mathbf{E}[ALG(\sigma)] \leq \mathcal{R} \cdot OPT(\sigma) \ , \tag{1}$$

gdzie wartość oczekiwana jest liczona po wszystkich wyborach losowych algorytmu.

Poniżej pokażemy, że współczynnik ten można poprawić stosując randomizację. Niech RAND będzie algorytmem, który losuje (z rozkładem jednostajnym) liczbę x z przedziału $[B/2, B]$ i czeka x ms na zwolnienie zasobu.

Twierdzenie 1.8. *Algorytm RAND jest $1,75$-konkurencyjny*

Dowód. Weźmy dowolne wejście I; niech T będzie momentem w którym zasób zostaje zwolniony. Oczywiście $\text{OPT}(I) = \min\{T, B\}$. Rozpatrzmy trzy przypadki.

1. $T > B$. Ponieważ zarówno OPT jak i RAND blokują zawsze przed czasem B, ich koszt nie zmieni się jeśli założymy, że $T = B$ (przypadek rozpatrzony w kolejnym punkcie).
2. $T < B/2$. W takim przypadku, $\text{RAND}(I) = T = \text{OPT}(I)$.
3. $B/2 \leq T \leq B$. Wtedy $\text{OPT}(I) = T$. Zauważmy, że gęstość rozkładu prawdopodobieństwa w przedziale $[B/2, B]$ to $2/B$. Niech $\text{RAND}(I, x)$ oznacza koszt algorytmu RAND na wejściu I pod warunkiem, że RAND wylosował x. Wtedy oczekiwany koszt algorytmu wynosi

$$\begin{aligned}
\mathbf{E}[\text{RAND}(\sigma)] &= \int_{B/2}^{B} \text{RAND}(I,x) \cdot \frac{2}{B}\, dx = \frac{2}{B} \cdot \left(\int_{B/2}^{T} \text{RAND}(I,x)\, dx + \int_{T}^{B} \text{RAND}(I,x)\, dx \right) \\
&= \frac{2}{B} \cdot \left(\int_{B/2}^{T} (x+B)\, dx + \int_{T}^{B} T\, dx \right) = \frac{2}{B} \cdot \left(\frac{T^2}{2} + B \cdot T - \frac{(B/2)^2}{2} - \frac{B^2}{2} + B \cdot T - T \cdot T \right) \\
&= \frac{2}{B} \cdot \left(2 \cdot B \cdot T - \frac{5}{8} \cdot B^2 - T^2/2 \right) = 4 \cdot T - \frac{5}{4} \cdot B - T^2/B \ .
\end{aligned}$$

Stąd wynika, że $\mathbf{E}[\text{RAND}(I)]/\text{OPT}(I) = 4 - \frac{5}{4} \cdot B/T - T/B$. Obliczając pochodną, otrzymujemy, że wyrażenie po prawej stronie maksymalizowane jest dla $T = \frac{\sqrt{5}}{2} \cdot B$. Jednak taka wartość jest poza przedziałem $[B/2, B]$ i dlatego współczynnik jest największy na jednym z końców (prawym) przedziału. Zatem

$$\frac{\mathbf{E}[\text{RAND}(I)]}{\text{OPT}(I)} \leq 4 - \frac{5}{4} \cdot B/B - B/B = 1,75 \ .$$

■

1.2. Eksploracja terenu

W.1B

WEJŚCIE NA PASTWISKO. Krowa stoi przy drodze w pewnej odległości do wejścia na pastwisko (większej od 1). Krowa jest ślepa i chce po omacku znaleźć wejście przechodząc jak najmniejszą odległość.

$d \leftarrow 1$

strona $\leftarrow$ *lewo*
repeat
  Idź d kroków w stronę *strona*
  Wróć do punktu startowego
  $d \leftarrow d \cdot 2$
  strona $\leftarrow$ *lewo*
until kwejście znalezione

Twierdzenie 1.9. *Powyższy algorytm jest ściśle* 9*-konkurencyjny.*

Dowód. Weźmy dowolną instancję wejściową σ. Niech x oznacza odległość wejścia od krowy. Oczywiście, $\textsc{Opt}(\sigma) = x$. Skoro $x > 1$ to istnieje liczba naturalna j spełniająca zależność

$$2^j < x \leq 2^{j+1} \ .$$

W najgorszym przypadku krowa posługująca się powyższym algorytmem w iteracji j przejdzie 2^j kroków w stronę wejścia, zawróci do punktu startowego, przejdzie 2^{j+1} w przeciwnym kierunku, zawróci do punktu startowego i w końcu przejdzie x kroków napotykając wejście na pastwisko. Całkowity koszt algorytmu wyniesie:

$$\begin{aligned} \textsc{Alg}(\sigma) &= 2 \cdot 1 + 2 \cdot 2 + 2 \cdot 4 + \ldots 2 \cdot 2^j + 2 \cdot 2^{j+1} + x \\ &= 2 \cdot (1 + 2 + \ldots + 2^{j+1}) + x \\ &\leq 2^{j+3} + x \leq 8 \cdot 2^j + x < 9 \cdot x = 9 \cdot \textsc{Opt}(\sigma) \ . \end{aligned}$$

■

1.3. Pakowanie pojemników

PAKOWANIE POJEMNIKÓW. Mamy nieskończony zbiór pojemników o rozmiarach równych 1. Ciąg wejściowy składa się z listy przedmiotów o wagach $w_i \leq 1$. Algorytm musi decydować, gdzie umieścić dany przedmiot, tak żeby suma wag w każdym pojemniku była co najwyżej 1. Celem jest minimalizacja liczby zużytych pojemników.

Algorytm FIRST-FIT (FF) wkłada przedmiot do pierwszego pojemnika, do którego włożenie jest możliwe.

Lemat 1.10. *Dla dowolnego wejścia* σ *zachodzi* $\textsc{First-Fit}(\sigma) \leq 2 \cdot \textsc{Opt}(\sigma) + 1$.

Dowód. Weźmy dowolny ciąg wejściowy σ zawierający m przedmiotów. Niech B oznacza zbiór wykorzystanych przez FIRST-FIT pojemników. Pokażemy, że we wszystkich pojemnikach, być może poza jednym, sumaryczna waga przedmiotów wynosi co najmniej 1/2. Załóżmy nie wprost, że są dwa takie pojemniki b_i i b_j (gdzie $i \leq j$) i rozważmy sytuację w momencie, w którym wkładany był pierwszy przedmiot do pojemnika b_j. Przedmiot ten miał wagę co najwyżej 1/2 a dodatkowo w pojemniku b_i było wtedy co najmniej 1/2 miejsca. Zatem przedmiot ten nie powinien był zostać włożony do „późniejszego" pojemnika b_j.

Niech k oznacza liczbę pojemników wykorzystanych przez algorytm FIRST-FIT. Bez straty ogólności załóżmy, że wszystkie pojemniki poza ostatnim mają zgromadzoną wagę przynajmniej 1/2. Zatem

$$\mathrm{FF}(\sigma) = k = 1 + (k-1) \leq 1 + \sum_{i=1}^{k-1} 2 \cdot w(b_i) \leq 1 + 2 \cdot \sum_{j=1}^{m} w(\sigma_j) \leq 1 + 2 \cdot \textsc{Opt}(\sigma) \ .$$

■

Powyższy wynik stanowi motywację do wprowadzenia konkurencyjności (nie-ścisłej).

Definicja 1.11. *Deterministyczny algorytm* ALG *jest* $\mathcal{R}$*-konkurencyjny jeśli istnieje stała* α*, taka że dla dowolnego wejścia* I *zachodzi*

$$\textsc{Alg}(I) \leq \mathcal{R} \cdot \textsc{Opt}(I) + \alpha \ . \tag{2}$$

W analogiczny sposób modyfikujemy definicję dla algorytmów zrandomizowanych.

Wniosek 1.12. *FIRST-FIT jest* 2*-konkurencyjny.*

2. Metoda zaznaczania: pamięć podręczna

W.2A
+5m

PAMIĘĆ PODRĘCZNA. Rozmiar RAM to N stron, rozmiar cache to k stron.
Wejście składa się z listy stron pamięci RAM, do których chce odwołać się procesor. Jeśli żądana strona X znajduje się już w pamięci cache, to koszt obsługi takiego żądania wynosi 0. W przeciwnym razie, algorytm *musi wczytać X do pamięci podręcznej* i koszt obsługi takiego żądania wynosi 1. Przed wczytaniem X do pamięci algorytm może wyrzucić dowolną liczbę stron z pamięci; jeśli pamięć podręczna przed wczytaniem jest pełna, to *musi* wyrzucić co najmniej jedną. Celem jest minimalizacja sumarycznego kosztu.

Powyższy model to model *w pełni asocjacyjny*. Jest jeszcze model *direct-mapping*, gdzie miejsce w cache na daną komórkę jest determinowane przez jej adres w RAM. W praktyce stosuje się mieszankę tych dwóch podejść. Istnieje wiele naturalnych i praktycznych algorytmów wykonujących to zadanie:

- LRU (*Least Recently Used*): wyrzuca stronę, do której odwołanie było najdalej w przeszłości
- FIFO (*First-In, First-Out*): wyrzuca stronę, która jest najdłużej w cache
- LFU (*Least Frequently Used*): wyrzuca stronę, która była najrzadziej używana
- FWF (*Flush When Full*): jeśli trzeba zrobić miejsce na stronę, wyrzuca wszystkie strony z pamięci podręcznej
- LFD (*Longest Forward Distance*): algorytm *offline*, który wyrzuca stronę do której następne odwołanie w ciągu znajduje się jak najpóźniej.

Zamiast analizować poszczególne algorytmy, skupimy się na ich pewnych cechach wspólnych.

2.1. Fazy

Aby udowodnić kolejne twierdzenie wprowadzimy najpierw definicję fazy. Ustalmy dowolną sekwencję wejściową σ. Faza 0 jest pustą sekwencją. Dla dowolnego $i \geq 1$, faza i jest maksymalną sekwencją następującą po fazie $i-1$, zawierającą co najwyżej k odwołań do *różnych* stron. Innymi słowy, faza $i+1$ (jeśli istnieje) zaczyna się na odwołaniu do $(k+1)$-szej różnej strony, licząc od początku fazy i. Zauważmy, że podział na fazy zależy tylko od sekwencji wejściowej.

Przykład dla $k = 3$, $N = 5$:

```
A B A C B | D A A B D | C E C A A C | B C
```

Co byśmy chcieli: płacić za każdą stronę co najwyżej raz w fazie. **Pomysł**: dla każdej strony utrzymuj jeden bit dodatkowych danych (zaznaczenie). przy odwołaniu do strony *zaznacz ją* (mark) i wyrzuć tę, która nie jest zaznaczona (dowolny podzbiór). Dokładniej:

wystąpiło odwołanie do strony X
if $X \notin$ cache **then**
  if cache jest pełny **then**
    if wszystkie strony zaznaczone **then**
      odznacz wszystkie strony
    wyrzuć dowolną niezaznaczoną stronę lub dowolne niezaznaczone strony
  wczytaj X do cache
zaznacz X

Mówimy o całej klasie algorytmów zaznaczających. Obserwacje:

- Strony poza pamięcią podręczną są zawsze niezaznaczone.
- Algorytm zaznaczający zaczyna fazę ze wszystkimi stronami w cache niezaznaczonymi i kończy ze zaznaczonymi.

2.2. Analiza algorytmu zaznaczającego

Obserwacja 2.1. *Dla dowolnej fazy f, $\textsc{Mark}(f) \leq k$.*

Ćwiczenie 2.2. *Pokaż, że $\textsc{Fwf}$ i $\textsc{Lru}$ są algorytmami zaznaczającymi.*

Obserwacja 2.3. *Jeśli f_i i f_{i+1} są dwoma kolejnymi fazami, to $\textsc{Opt}(f_i \cup f_{i+1}) \geq 1$.*

Dowód. W ciągu $f_i \cup f_{i+1}$ znajdują się odwołania do co najmniej $k+1$ różnych faz. Zatem $\textsc{Opt}$ musi podczas takiego ciągu zapłacić co najmniej raz. ■

Z Obserwacji 2.1 i Obserwacji 2.3 można pokazać, że $\textsc{Mark}$ jest $2k$-konkurencyjny. Chcielibyśmy poprawić trochę tę analizę.

Wyobraźmy sobie fazę i jako okienko o pewnej długości przez które widać pewien ciąg odwołań do pamięci i przesuńmy to okienko o jedno odwołanie w prawo; nowy ciąg nazywamy i-tą *fazą przesuniętą*. Innymi słowy i-ta faza przesunięta zaczyna się na drugim odwołaniu z fazy i i kończy na pierwszym odwołaniu z fazy $i+1$ (włącznie).

Lemat 2.4. *Dla dowolnej fazy przesuniętej (poza być może ostatnią) f'_i zachodzi $\textsc{Opt}(f'_i) \geq 1$.*

Dowód. Niech q będzie stroną, do której jest odwołanie w pierwszym kroku f_i. Wtedy każdy algorytm (także $\textsc{Opt}$) ma q w swojej pamięci podręcznej na początku f'_i. f'_i musi zawierać k odwołań do różnych stron (dodatkowo różnych od q). Zatem w ciągu fazy przesuniętej $\textsc{Opt}$ co najmniej raz musi załadować stronę z pamięci RAM. ■

Twierdzenie 2.5. *Dowolny algorytm zaznaczający $\textsc{Mark}$ jest k-konkurencyjny.*

Dowód. Weźmy dowolną sekwencję wejściową σ i jej podział na fazy. Załóżmy że tych faz jest $\ell+1$; ostatnia faza jest być może nie zakończona. Wtedy $\textsc{Mark}(\sigma) \leq (\ell+1) \cdot k$ oraz $\textsc{Opt}(\sigma) \geq \ell \cdot 1$. Stąd $\textsc{Mark}(\sigma) \leq k \cdot \textsc{Opt}(\sigma) + k$. ■

2.3. Dolne ograniczenie

Addytywny składnik w definicji konkurencyjności powoduje mały problem w dowodzeniu dolnych ograniczeń na konkurencyjność. Nie wystarczy już istnienie pojedynczego wejścia na którym algorym ma duży koszt a algorytm optymalny ma koszt odpowiednio mały; trzeba pokazać, że taka relacja zachodzi dla wejść o dowolnie dużym koszcie.

Lemat 2.6. *Ustalmy dowolny deterministyczny algorytm $\textsc{Det}$. Jeśli istnieje nieskończony ciąg wejść $\sigma_1, \sigma_2, \ldots$ taki że:*

1. $\lim_{n\to\infty} \textsc{Det}(\sigma_n)/\textsc{Opt}(\sigma_n) \geq R$ *oraz*
2. $\lim_{n\to\infty} \textsc{Det}(\sigma_n) = \infty$,

to R jest dolnym ograniczeniem na konkurencyjność $\textsc{Det}$.

Dowód. Możemy założyć, że $\lim_{n\to\infty} \textsc{Opt}(\sigma_n) = \infty$; w przeciwnym przypadku $\textsc{Det}$ nie jest w ogóle konkurencyjny.

Załóżmy nie wprost, że $\textsc{Det}$ jest $(R-\varepsilon)$-konkurencyjny. Wtedy istnieje takie α, że dla dowolnego wejścia σ zachodzi

$$\textsc{Det}(\sigma) \leq (R-\varepsilon) \cdot \textsc{Opt}(\sigma) + \alpha \ . \tag{3}$$

Z ciągu wejść $(\sigma_i)_i$ wybierzmy wszystkie wejścia σ'_i dla których $\textsc{Opt}(\sigma'_i) \geq 2\alpha/\varepsilon$. Tworzą one nieskonczony ciąg

wejść $(\sigma'_i)_i$. Biorąc dowolne z nich i podstawiając do (3) otrzymujemy

$$\textsc{Det}(\sigma'_i)/\textsc{Opt}(\sigma'_i) \leq (R-\varepsilon) + \alpha/\textsc{Opt}(\sigma'_i) \leq R - \varepsilon + \varepsilon/2 = R - \varepsilon/2 \ .$$

Istnienie takiego nieskończonego podciągu $(\sigma_i)_i$ przeczy warunkowi $\lim_{n\to\infty} \textsc{Det}(\sigma_n)/\textsc{Opt}(\sigma_n) \geq R$. ■

Dla dowodu dolnego ograniczenia zakładamy N, rozmiar pamięci RAM, jest większy niż k. Będziemy korzystać z pierwszych $k+1$ stron pamięci RAM. Zakładamy również, że na początku w pamięci podręcznej są strony $1, 2, \ldots, k$. Zacznijmy od następującego lematu.

Lemat 2.7. *Jeśli* LFD *rozpoczyna z pełną pamięcią podręczną, to dla dowolnej sekwencji σ złożonej z odwołań do $k+1$ różnych stron zachodzi*

$$\textsc{Lfd}(\sigma) \;\leq\; 1 + |\sigma|/k \;\;,$$

gdzie LFD *jest algorytmem (offline) Longest Forward Distance.*

Dowód. Ustalmy dowolne odwołanie w sekwencji σ, takie, że algorytm LFD usunął jakąś stronę (oznaczmy ją Q) z pamięci podręcznej. Z definicji algorytmu LFD wynika, że spośród wszystkich stron, które przed chwilą były w pamięci podręcznej odwołanie do Q było najpóźniej. Innymi słowy zanim przyjdzie pierwsze odwołanie do Q, procesor odwoła się do wszystkich innych $k-1$ stron z cache. Dopiero odwołanie do Q powoduje koszt, gdyż pozostałe k stron mamy w pamięci podręcznej. Zatem odwołanie do pamięci RAM pojawia się najwyżej raz na k kroków. ■

Twierdzenie 2.8. *Dolne ograniczenie na konkurencyjność dowolnego algorytmu deterministycznego dla problemu pamięci podręcznej wynosi k.*

Dowód. Ustalmy dowolny deterministyczny algorytm ALG. Bez straty ogólności możemy założyć, że na początku w pamięci podręcznej są strony $1, 2, \ldots, k$.[1] Konstrukcja złośliwego ciągu wejściowego jest prosta — adwersarz zawsze żąda dokładnie tej strony, której ALG nie ma w pamięci podręcznej. Oczywiście taka sekwencja wejściowa może być dowolnie długa (i dowolnie kosztowna). Ponieważ ALG płaci wtedy za każde odwołanie do strony, $\textsc{Alg}(\sigma) = |\sigma|$. Łącząc to z Lematem 2.7 otrzymujemy

$$\textsc{Alg}(\sigma) = |\sigma| \leq k \cdot \textsc{Lfd}(\sigma) - k \;\geq\; k \cdot \textsc{Opt}(\sigma) - k \;\;.$$ ■

2.4. Randomizacja algorytmu zaznaczającego

Algorytm R-MARK jest prostym losowym algorytmem markującym, który gdy trzeba zrobić miejsce w pamięci podręcznej usuwa z niej losową niezaznaczoną stronę. Każda niezaznaczona strona zostaje wybrana z jednakowym prawdopodobieństwem. W.2B

Twierdzenie 2.9. *Algorytm R-*MARK *jest $2 \cdot H_k$-konkurencyjny.*

Dowód. Dowolny algorytm zaznaczający zaczyna każdą fazę ze wszystkimi stronami niezaznaczonymi i kończy ją ze wszystkimi zaznaczonymi. Tak jest również w przypadku algorytmu randomizowanego. **Strony, które są zaznaczone nie zależą od randomizacji.**

Rozpatrzmy dowolną fazę f_i i przyjrzyjmy się tym stronom do których występuje odwołanie w f_i. Możemy ograniczyć się do analizy tylko pierwszego wystąpienia danej strony (potem strona zostaje zaznaczona i nie zostanie wyrzucona aż do końca fazy, zatem wszystkie następne odwołania nie generują kosztu). Strony dzielimy na trzy zbiory:

1. Strony nowe $= f_i \setminus f_{i-1}$. (Ich liczba to n_i).
2. Strony stare $= f_i \cap f_{i-1}$.
3. Strony bezużyteczne $= f_{i-1} \setminus f_i$.

Jaki jest koszt $\textsc{Opt}(\sigma)$? W fazie pierwszej OPT płaci co najmniej n_1. Rozważmy dowolną fazę $i > 1$. W ciągu złożonym z fazy $i-1$ i fazy i występuje $k + n_i$ odwołań do różnych stron, zatem OPT chybia tam co najmniej n_i razy. Jeśli zsumujemy wszystkie takie ciągi zawarte w σ i pierwszą fazę, to sumaryczny koszt wynosi $\sum_i n_i$ a każda faza (poza ostatnią) jest liczona podwójnie. Dlatego też

$$\textsc{Opt}(\sigma) \geq \frac{1}{2} \cdot \sum_i n_i \;\;.$$

Wystarczy pokazać, że $\mathbf{E}[\text{R-}\textsc{Mark}(\sigma)] \leq \sum_i n_i \cdot H_k$. Ustalmy dowolną fazę i. Każda z nowych stron to koszt 1. Jaki jest koszt starych stron? Oznaczmy je przez $S_1, S_2, \ldots, S_{k-n_i}$.

$$\mathbf{E}[\text{R-}\textsc{Mark}(f_i)] = n_i + \sum_{j=1}^{k-n_i} \mathbf{Pr}\left[\,\text{przy pierwszym odwołaniu do strony } S_j \text{ nie ma jej w cache}\,\right] \;\;.$$

Jak wyglądał cache na samym początku fazy? Mieliśmy w nim:

[1] Jeśli pamięć podręczna jest na początku pusta, to możemy dodać do sekwencji prefiks ładujący te strony, powodujący stały koszt k zarówno u ALG jak i u OPT. Ten stały koszt przestanie mieć znaczenie przy odpowiednio kosztownej pozostałej części sekwencji.

- Strony stare: $S_1, S_2, \ldots, S_{k-n_i}$.
- Strony bezużyteczne: $B_1, B_2, \ldots, B_{n_i}$.

Idealna sytuacja: najpierw odwołania do starych, potem do nowych stron. Najgorsza sytuacja: najpierw nowe strony, potem stare.

Jak wygląda cache na moment przed odwołaniem do S_j? Mamy w nim:

- Strony stare: $S_1, S_2, \ldots, S_{j-1}$.
- Pewną liczbę stron nowych: $N_1, N_2, \ldots, N_{h(j)}$.
- Co jeszcze? W pozostałych $k-(j-1)-h(j)$ slotach mamy pewne strony ze zbioru $\{S_j, S_{j+1}, \ldots, S_{k-n_i}\} \cup \{B_1, B_2, \ldots, B_{n_i}\}$, przy czym ze względu na symetrię każdy podzbiór jest jednakowo prawdopodobny.

Jaka jest szansa, że w tych pozostałych $P := k - j + 1 - h(j)$ slotach mamy konkretną stronę S_j ze zbioru o $Q := k - n_i - (j-1) + n_i = k - j + 1$ elementach? Jest ona równa P/Q, bo:

1. Wszystkich wyborów P-elementowych podzbiorów z Q-elementowego zbioru jest $\binom{Q}{P}$, zaś tych, które zawierają S_j jest $\binom{Q-1}{P-1}$ i stąd
$$\frac{\binom{Q-1}{P-1}}{\binom{Q}{P}} = \frac{P}{Q} \ .$$
2. Albo inaczej: konkretna strona s występuje z (takim samym) prawdopodobieństwem p. Definiujemy zmienną losową X_s będącą indykatorem zdarzenia „s jest w zbiorze P”. Wtedy $E[X_s] = p$. Z drugiej strony $\sum_s X_s = P$ dla dowolnego losowania. A zatem
$$P = \mathbf{E}[\sum_s X_s] = \sum_s \mathbf{E}[X_s] = Q \cdot p$$

Stąd mamy

$$\begin{aligned}
\mathbf{E}[\text{R-MARK}(f_i)] &= n_i + \sum_{j=1}^{k-n_i} \left(1 - \frac{k-j+1-h(j)}{k-j+1)}\right) \\
&= n_i + \sum_{j=1}^{k-n_i} \frac{h(j)}{k-j+1} \leq n_i + \sum_{j=1}^{k-n_i} \frac{n_i}{k-j+1} \\
&\leq n_i + \sum_{j=1}^{k-1} \frac{n_i}{k-j+1} = n_i + \sum_{j=2}^{k} \frac{n_i}{j} = n_i \cdot H_k \ .
\end{aligned}$$

Sumujące ten koszt po wszystkich fazach dostaniemy tezę twierdzenia. ■

3. Zasada zaznaczania: MTS. Funkcje potencjału: reorganizacja listy

3.1. MTS

W.3A
+10m

Metryczne systemy zadań. Dany jest skończony zbiór stanów S i metryka d na tym zbiorze. Dany jest również stan początkowy $s_0 \in S$. W każdym kroku t algorytm otrzymuje wektor kar r_t, określający ile trzeba zapłacić za bycie w danym stanie. Algorytm zmienia swój stan z s_{t-1} do s_t płacąc $d(s_{t-1}, s_t)$ (możliwe, że $s_t = s_{t-1}$) a następnie płaci $r_t(s_t)$. Należy zminimalizować sumaryczny koszt.

Obserwacja 3.1. *SRP to MTS*

Obserwacja 3.2. *Problem pamięci podręcznej to MTS*

Algorytm zaznaczający dla metryki uniform (koszt zmiany stanu = D) i jednostkowych kosztów ($r_t \in \{0,1\}^n$), gdzie $n = |S|$: Działamy w epokach. Na końcu kroku, w którym suma kar w epoce w stanie s_i przekracza D zaznaczamy stan s_i. Jeśli algorytm jest w takim wierzchołku to zmienia stan na dowolny niezaznaczony. Jeśli wszystko staje się zaznaczone, to kończymy epokę i odznaczamy wszystkie stany.

Przykład epoki dla $D = 3$

```
 |s|s|s|s|s|s|
 |1|2|3|4|5|6|
---------------
 |1| |1| | | |
 | |1|1| |1|1|
 |1|1| | |1| |
 | |X| |1| |1|
 |X|1| | | |X|
 | |1|X|1|X|1|
 |1| | |X| | |
--------------
```

Twierdzenie 3.3. *Algorytm zaznaczający jest $O(n)$-konkurencyjny.*

W dowodzie pokazujemy, że:

1. Dla dowolnej epoki E, $\textsc{Opt}(E) \geq D$.
2. Algorytm płaci co najwyżej $n \cdot D$ (za przenosiny) i $n \cdot D$ (za obsługę żądań).

3.2. Reorganizacja listy

Mamy daną listę jednokierunkową ze wskaźnikiem na początek tej listy. Załóżmy, że lista ma w danej chwili długość ℓ. Definiujemy na liście operację $\mathsf{Access}(x)$, która kosztuje i jeśli x jest na miejscu i-tym w liście.

Reorganizacja listy. Wejście dla problemu składa się z sekwencji operacji Access. Bezpośrednio po operacji $\mathsf{Access}(x)$, 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.

Płatne zamiany: my nie będziemy tego robić, OPT może.
Są trzy naturalne algorytmy: Move-To-Front, Transpose i Frequency Count. Który jest najlepszy?

3.3. Dolne ograniczenie: argument o średniej

Twierdzenie 3.4. *Współczynnik konkurencyjności dowolnego algorytmu deterministycznego wynosi co najmniej* $2 - 2/(\ell + 1)$.

Dowód. Ustalmy dowolny algorytm deterministyczny DET i dowolną listę początkową o długości ℓ. Ustalmy dowolne m, które będzie długością sekwencji wejściowej σ. Wszystkie żądania będą dotyczyć ostatniego elementu na liście. Zatem $\text{DET}(\sigma) = \ell \cdot m$.

Do ograniczenia OPT wykorzystamy $\ell!$ „statycznych" algorytmów STAT_π. Taki algorytm ustala na samym początku permutację π elementów na liście i nigdy go potem nie zmienia. Maksimum ich kosztów związanych z początkową reorganizacją listy oznaczmy przez b; $b = \mathcal{O}(\ell^2)$. Mamy

$$\sum_\pi \text{STAT}_\pi(\sigma) \le \sum_\pi \left(b + \sum_{t=1}^{m} \text{STAT}_\pi(\sigma_t) \right) = b \cdot \ell! + \sum_{t=1}^{m} \sum_\pi \text{STAT}_\pi(\sigma_t) \ .$$

Czym jest $\sum_\pi \text{STAT}_\pi(\sigma_t)$? W kroku t $(\ell-1)!$ algorytmów ma σ_t na pierwszej pozycji, $(\ell-1)!$ algorytmów ma σ_t na drugiej pozycji itd. Stąd $\sum_\pi \text{STAT}_\pi(\sigma_t) = (\ell-1)! \cdot \ell \cdot (\ell+1)/2$. Oczywiście OPT jest nie gorszy niż najtańsze rozwiązanie STAT_π, a zatem

$$\text{OPT}(\sigma) \le \min_\pi \ \text{STAT}_\pi(\sigma) \le \frac{\sum_\pi \text{STAT}_\pi(\sigma)}{\ell!} \le b + m \cdot (\ell+1)/2 \ .$$

Otzymujemy zatem $\lim_{m\to\infty} \text{DET}(\sigma)/\text{OPT}(\sigma) = 2\ell/(\ell+1) = 2 - 2/(\ell+1)$. ■

3.4. Algorytm Move-To-Front

W tej części rozważamy Algorytm MTF (MOVE-TO-FRONT), który po każdym żądaniu σ_t przenosi σ_t (za darmo) na początek listy. W.3B

Twierdzenie 3.5. *Algorytm MTF jest ściśle* 2*-konkurencyjny.*

Jaka jest idea dowodów z potencjałem?

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 $1 \le t \le m$ zachodzi

$$\text{MTF}(t) + \Phi(t) - \Phi(t-1) \le 2 \cdot \text{OPT}(t) \tag{4}$$

Jeśli zsumujemy powyższą nierówność po wszystkich krokach t otrzymamy $\text{MTF}(\sigma) + \Phi(m) - \Phi(0) \le 2 \cdot \text{OPT}(\sigma)$, czyli $\text{MTF}(\sigma) \le 2 \cdot \text{OPT}(\sigma)$.

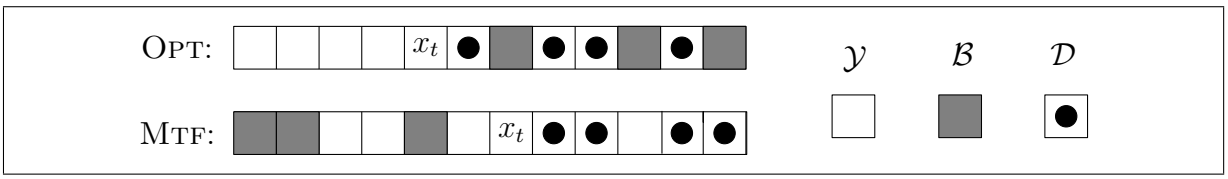
Zauważmy, że jeśli nie wymagamy ścisłej konkurencyjności, powyższy rezultat zachodzi nawet gdy MTF i OPT startują z różnych ustawień początkowych; w takim przypadku $\Phi(0) \neq 0$, ale możemy umieścić $\Phi(0)$ w stałej addytywnej z definicji konkurencyjności.

W pozostałej części dowodu pokażemy nierówność (4). 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 (4) zachodzi.



Można zrobić tutaj dokładniejszą analizę, co się dzieje z inwersjami różnego typu.

Dowód dla pierwszej akcji. Niech A będzie liczbą elementów, które poprzedzają σ_t w liście MTF i liście OPT. Niech B będzie liczbą elementów, które poprzedzają σ_t w liście MTF, ale występują po σ_t w liście OPT. Wtedy mamy

$$\text{MTF}(t) \;=\; A + B + 1 \;, \qquad \text{OPT}(t) \;\geq\; A + 1 \;.$$

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

$$\text{MTF}(t) + \Delta\Phi(t) \;\leq\; 2A + 1 \;\leq\; 2 \cdot \text{OPT}(t) - 1$$

Udowodniliśmy, że (4) zachodzi też dla pierwszej akcji.

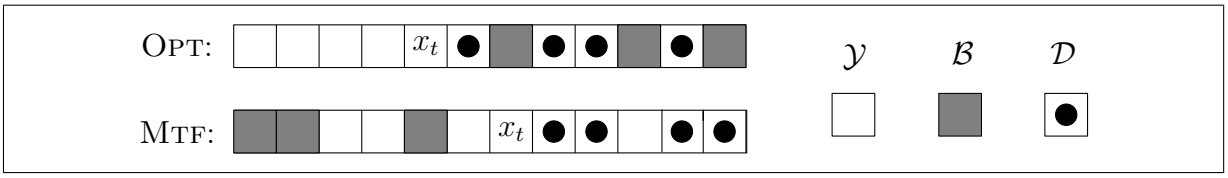
Dowód dla drugiej akcji. Niech P oznacza liczbę płatnych zamian wykonywanych przez algorytm OPT, tj. $\text{OPT} = P$. Każda taka zamiana sąsiednich elementów powoduje wzrost potencjału o co najwyżej 1. Zatem całkowity wzrost potencjału $\Delta\Phi$ związany z tą akcją jest ograniczony z góry przez koszt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

4. Funkcje potencjału dla algorytmów randomizowanych

4.1. Reorganizacja listy: algorytm BIT

Idea modyfikacji algorytmu MTF: Analiza algorytmu MTF zakłada najgorszy dla nas przypadek, w których OPT może nie przemieścić elementu σ_t i dzięki temu nasz potencjał zwiększa się dodatkowo o A. Z drugiej strony algorytm, który nie będzie się ruszać w ogóle, będzie niekonkurencyjny. Pomysł polega więc na tym, żeby wykorzystać randomizację do lepszego „łapania" OPT-a, tj. przesuwać element z prawdopodobieństwem 1/2.

Przypomnijmy teraz związany z MTF rysunek zastępując zbiory $\mathcal{A}$ i $\mathcal{C}$ jednym zbiorem nazwanym $\mathcal{Y}$.



Na początku algorytm BIT dla każdego elementu x wybiera losowo $b(x) \in \{0, 1\}$. Nastepnie podczas obsługi odwołania do elementu x wykonaj:

FLIP($b(x)$)
if $b(x) = 1$ then przesuń x na początek listy

Zauważmy, że algorytm BIT jest używa stałej liczby bitów losowych (ℓ), żeby obsłużyć sekwencje wejściową o dowolnej długości. Z faktu, że $b(x)$ jest losowane na początku i trzymane w tajemnicy przed adwersarzem otrzymujemy następujący fakt.

Obserwacja 4.1. *Zauważmy, że w danym kroku t, dla dowolnego x, $b(x)$ jest zmienną losową przyjmującą wartość 1 z prawdopodobieństwem 1/2 i 0 z takim samym prawdopodobieństwem, niezależnie od liczby odwołań do x i stanu listy algorytmu optymalnego.*

Musimy zmodyfikować dodatkowo funkcję potencjału: w przypadku kroku t, takiego że $b(\sigma_t) = 1$, BIT musi potencjałem skompensować to, że OPT może mieć koszt o B mniejszy od BIT.

Twierdzenie 4.2. *Algorytm BIT jest* 1,75*-konkurencyjny.*

Dowód. Wybierzmy dowolną sekwencję σ. Uruchomimy na niej algorytm BIT i algorytm optymalny OPT i będziemy obserwować jak się na niej zachowują.

Do dowodu będziemy potrzebować paru oznaczeń. Niech $x \prec_{\text{alg}} y$ oznacza, że x występuje przed y (x jest bliżej początku listy niż y) na liście algorytmu ALG. *Inwersją* nazywamy uporządkowaną parę (x, y), taką że $x \prec_{\text{opt}} y$ oraz $y \prec_{\text{bit}} x$. Mówimy, że inwersja (x, y) jest typu $b(x)$. Niech ϕ_i oznacza liczbę inwersji typu i. Definiujemy funkcję potencjału jako

$$\Phi = \phi_0 + 2 \cdot \phi_1 \ .$$

Rozważmy krok t w którym występuje odwołanie do elementu σ_t. Dla dowodu konkurencyjności algorytmu wystarczy pokazać, że

$$\mathbf{E}[\text{BIT}(t)] + \mathbf{E}[\Delta\Phi(t)] \ \leq \ (7/4) \cdot \text{OPT}(t) \ . \tag{5}$$

Podobnie jak w dowodzie konkurencyjności algorytmu MTF podzielimy krok t na dwie akcje. W pierwszej akcji BIT i OPT płacą za żądanie $\mathsf{Access}(\sigma_t)$, BIT wykonuje swoje (bezpłatne) zamiany a OPT wykonuje swoje bezpłatne zamiany. W drugiej akcji OPT wykonuje swoje płatne zamiany i płaci za nie. Udowodnimy, że nierówność (5) zachodzi dla każdej z akcji z osobna.

Dowód nierówności (5) dla pierwszej akcji. Niech $Y + 1$ będzie pozycją na której σ_t jest w liście OPT. Niech B będzie zmienną losową oznaczającą liczbę inwersji (σ_t, y). Typ tych inwersji zależy oczywiście od $b(\sigma_t)$. Wtedy OPT $= Y + 1$ zaś BIT $\leq Y + B$. OPT przesuwa się o $Y - Y'$ pozycji do przodu na liście. Pozostaje zatem ograniczyć zmianę potencjału.

- Załóżmy, że na początku kroku t, $b(\sigma_t) = 1$.

  Wtedy BIT zmienia wartość $b(\sigma_t)$ na 0 i nie przesuwa σ_t. Każda inwersja (σ_t, y) zmienia wtedy typ z 1 na 0, a związana z tym zmiana potencjału to $-B$. OPT może następnie przesunąć σ_t w stronę początku listy. Wpływa to na status par $(y_{Y'+1}, \sigma_t), (y_{Y'+2}, \sigma_t), \ldots, (y_Y, \sigma_t)$: te z nich które były inwersjami przestają nimi być, zaś te, które nie były inwersjami, stają się nimi. Wszystkie tworzone w ten sposób inwersje mają typ 0, a zatem związana z tym zmiana potencjału to co najwyżej $Y - Y'$. Otrzymujemy zatem $\mathbf{E}[\Delta\Phi'' | b(\sigma_t) = 0] \leq -B + Y - Y'$.

- Załóżmy, że na początku kroku t, $b(\sigma_t) = 0$.

  Wtedy BIT zmienia wartość $b(\sigma_t)$ na 1 i przesuwa σ_t na początek. Usuwa to wszystkie I inwersji (σ_t, b) (o typie 0). Związana z tym zmiana potencjału to $-B$.

  Po zmianie bitu, $b(\sigma_t) = 1$ i σ_t jest przesuwany na początek. Podobnie jak poprzednio, wszyscy kandydaci na nowe inwersje są na liście par $(y_1, \sigma_t), (y_2, \sigma_t), \ldots, (y_{Y'}, \sigma_t)$ Każda z tych inwersji jest z jednakowym prawdopodobieństwem (zależnym od bitu $b(y_i)$) typu 0 i typu 1. Dlatego $\mathbf{E}[\Delta\Phi''|b(\sigma_t) = 1] \leq (Y' - 1) \cdot (\frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 2) < \frac{3}{2} \cdot Y'$.

Po połączeniu obu przypadków otrzymujemy

$$\mathbf{E}[\Delta\Phi''] = \tfrac{1}{2} \cdot \mathbf{E}[\Delta\Phi''|b(\sigma_t) = 0] + \tfrac{1}{2} \cdot \mathbf{E}[\Delta\Phi''|b(\sigma_t) = 1] \leq \tfrac{1}{2} \cdot Y - \tfrac{1}{2} \cdot Y' + \tfrac{3}{4} \cdot Y' \leq \tfrac{3}{4} \cdot Y \ .$$

Zatem ostatecznie

$$\mathbf{E}[\text{BIT} + \Delta\Phi] = \mathbf{E}[\text{BIT} + \Delta\Phi' + \Delta\Phi''] \leq \tfrac{7}{4} \cdot Y = \tfrac{7}{4} \cdot \text{OPT} \ .$$

Dowód nierówności (5) dla drugiej akcji. Oczywiście BIT = 0. Dla każdej zamiany wykonywanej przez OPT związany z nią koszt wynosi 1 i tworzona jest wtedy co najwyżej jedna inwersja. Prawdopodobieństwo, że jej typ to 1 lub 2 wynosi 1/2. Zatem oczekiwany wzrost potencjału to co najwyżej

$$\mathbf{E}[\Delta\Phi(t)] \ = \ \tfrac{1}{2} \cdot 1 + \tfrac{1}{2} \cdot 2 \ = \ \tfrac{3}{2} \ \leq \ \tfrac{3}{2} \cdot \text{OPT}(t) \ . \qquad \blacksquare$$

4.2. Pamięć podręczna: algorytm RAND

Rozważmy następujący losowy algorytm pamięci podręcznej RAND. Jeśli w ciągu wejściowym występuje odwołanie do strony, której nie ma w pamięci podręcznej, to RAND wyrzuca z pamięci podręcznej losową stronę. Udowodnimy, że współczynnik konkurencyjności algorytmu RAND przeciwko adwersarzowi nieświadomemu i adwersarzowi adaptującemu się wynosi k. W tym celu udowodnimy następujące dwa twierdzenia.

Twierdzenie 4.3. *RAND jest k-konkurencyjny przeciwko adwersarzowi adaptującemu się.*

Dowód. Zdefiniujemy funkcję potencjału, która będzie tym większa, im większa jest różnica między tym co w pamięci podręcznej trzyma RAND a tym co trzyma adwersarz OPT. Bez straty ogólności możemy założyć, że algorytm stosowany przez OPT wyrzuca w każdym kroku co najwyżej jedną stronę. Niech w dowolnej chwili P będzie zbiorem wszystkich stron, które są zarówno w pamięci podręcznej RAND jak i OPT. Stronę należącą do tego zbioru nazywamy *wspólną*. Definiujemy funkcję potencjału jako

$$\Phi = k \cdot (k - |\mathcal{P}|) \ .$$

Oczywiście Φ jest zawsze dodatnie. Wystarczy że pokażemy, że dla dowolnych stanów pamięci podręcznych RAND i OPT na początku jakiegoś kroku i dla odwołania do dowolnej strony, zachodzi

$$\mathbf{E}[\text{RAND}] + \mathbf{E}[\Delta\Phi] \leq k \cdot \mathbf{E}[\text{OPT}] \ . \tag{6}$$

Sumując to po wszystkich krokach, korzystając z liniowości wartości oczekiwanej i nieujemności funkcji potencjału dostajemy, $\mathbf{E}[\text{RAND}(\sigma)] \leq k \cdot \mathbf{E}[\text{OPT}(\sigma)] + \Phi_0$, co dowodzi konkurencyjności algorytmu.

Na marginesie zauważmy, że w przypadku dowodu z funkcją potencjału dla algorytmu BIT walczącego przeciwko adwersarzowi oblivious, stan algorytmu był zmienną losową niezależną od adwersarza. W przypadku adwersarza adaptującego się, w każdym kroku musimy rozpatrywać dowolny możliwy stan pamięci podręcznej RAND.

Wystarczy zatem udowodnić (6). W tym celu ustalmy dowolny krok t i oznaczmy stronę do której jest odwołanie w tym kroku przez p. Rozważamy cztery przypadki:

Przypadek 1 *Strona p jest w pamięci podręcznej RAND.* Wtedy RAND = 0. Jeśli adwersarz ma p w pamięci podręcznej, OPT = $\Delta\Phi$ = 0. W przeciwnym przypadku OPT = 1, a $\Delta\Phi \leq k$ (bo $\mathcal{P}$ może się zmienić co najwyżej o jeden element).

Przypadek 2 *Strony p nie ma w pamięci podręcznej RAND, ale jest w pamięci podręcznej OPT.* Wtedy RAND = 1 i OPT = 0, zatem wystarczy pokazać, że $\mathbf{E}[\Delta\Phi] \leq -1$. Na początku tego ruchu $|\mathcal{P}| \leq k - 1$. Z prawdopodobieństwem $|P|/k$, algorytm wyrzuca ze swojej pamięci wspólną stronę. Wtedy nie ma zmiany w liczności zbioru P, bo RAND wrzuca do pamięci wspólną stronę p. W przeciwnym przypadku zbiór P zwiększa się o 1 i potencjał maleje o k. Zatem

$$\mathbf{E}[\Delta\Phi] = \frac{|\mathcal{P}|}{k} \cdot 0 + \frac{(k - |\mathcal{P}|)}{k} \cdot (-k) \leq -\frac{1}{k} \cdot k = -1 \ .$$

Przypadek 3 *Strony p nie ma w pamięciach podręcznych* RAND *ani* OPT*, a adwersarz wyrzuca niewspólną stronę.* Wtedy RAND = OPT = 1. W tym przypadku zbiór $\mathcal{P}$ nie może się zmniejszyć (a pozostaje taki sam jeśli RAND wyrzuci wspólną stronę), a zatem $\Delta\Phi \leq 0$.

Przypadek 4 *Strony p nie ma w pamięciach podręcznych* RAND *ani* OPT*, a adwersarz wyrzuca wspólną stronę $q \in \mathcal{P}$.* Jak w poprzednim przypadku RAND = OPT = 1. Jeśli algorytm wyrzuci stronę q albo dowolną stronę nie będącą w $\mathcal{P}$, to liczność zbioru $\mathcal{P}$ a więc i potencjał nie zmienia się. Jeśli natomiast algorytm wyrzuci stronę należącą do $P \setminus \{q\}$ (co zdarza się z prawdopodobieństwem $\frac{|\mathcal{P}|-1}{k}$, to potencjał zwiększy się o k. Wtedy

$$\mathbf{E}[\Delta\Phi] = \frac{|\mathcal{P}|-1}{k} \cdot k = |\mathcal{P}| - 1 \leq k - 1 \ .$$

Zatem w każdym z powyższych przypadków zachodzi nierówność 6. ■

5. Dolne granice dla algorytmów randomizowanych: zasada minimaksowa

5.1. Wariant dla ścisłej konkurencyjności

Lemat 5.1 (Zasada minimaksowa Yao, wersja dla ścisłej konkurencyjności)**.** *Rozważmy dowolny problem minimalizacyjny. Wybierzmy dowolny rozkład prawdopodobieństwa π nad zbiorem wszystkich możliwych sekwencji wejściowych $\mathcal{I}$. Jeśli dla dowolnego deterministycznego algorytmu* DET *zachodzi*

$$\mathbf{E}_\pi[DET(\sigma)] \geq \mathcal{R} \cdot \mathbf{E}_\pi[OPT(\sigma)] \ ,$$

to $\mathcal{R}$ jest dolnym ograniczeniem na ścisłą konkurencyjność dowolnego zrandomizowanego algorytmu.

Najpierw przykład: zastosowanie do problemu wypożyczania nart.

Przykład 5.2. *Załóżmy, że $3|B$. Wtedy dolne ograniczenie na konkurencyjność dowolnego algorytmu zrandomizowanego dla problemu wypożyczania nart wynosi co najmniej $6/5$.*

Dowód. Rozkład prawdopodobieństwa π jest następujący: z prawdopodobieństwem $2/3$ narciarz łamie nogę dnia $B/3$ (wejście σ_1) a z prawdopodobieństwem $1/3$ nie łamie nogi nigdy (wejście σ_2).

Wtedy $\textsc{Opt}(\sigma_1) = B/3$ i $\textsc{Opt}(\sigma_2) = B$, a zatem $\mathbf{E}_\pi[\textsc{Opt}(I)] = \frac{2}{3} \cdot \textsc{Opt}(\sigma_1) + \frac{1}{3} \cdot \textsc{Opt}(\sigma_2) = \frac{5}{9}B$.

Jaki jest najlepszy algorytm deterministyczny dla takiego rozkładu π? Zauważmy, że na końcu dnia $B/3$ algorytm deterministyczny „wie" już, czy ma do czynienia z wejściem σ_1 czy σ_2. Dodatkowo jeśli algorytm decyduje się na kupno nart dnia $B/3$ lub wcześniej, lepiej postąpi jeśli kupi je od razu dnia pierwszego. Zatem jedyne sensowne algorytmy deterministyczne to $\textsc{Det}_1$ kupujący narty dnia 1 i $\textsc{Det}_2$ kupujący narty dnia $B/3+1$ (pod warunkiem, że nogi są jeszcze niepołamane).

Mamy $\mathbf{E}_\pi[\textsc{Det}_1(\sigma)] = B$ oraz $\mathbf{E}_\pi[\textsc{Det}_2(I)] = \frac{2}{3} \cdot \textsc{Det}_2(\sigma_1) + \frac{1}{3} \cdot \textsc{Det}_2(\sigma_2) = \frac{2}{3} \cdot B/3 + \frac{1}{3} \cdot (B/3 + B) = \frac{6}{9}B$. Zatem oczekiwany koszt najlepszego algorytmu deterministycznego DET wynosi $\frac{6}{9}B$. Stąd dolne ograniczenie na konkurencyjność dowolnego algorytmu randomizowanego wynosi co najmniej

$$\frac{\mathbf{E}_\pi[\textsc{Det}(I)]}{\mathbf{E}_\pi[\textsc{Opt}(I)]} = \frac{\frac{6}{9} \cdot B}{\frac{5}{9} \cdot B} = \frac{6}{5} \ . \qquad \blacksquare$$

Dowód zasady minimaksowej. Ustalmy dowolny algorytm randomizowany ALG. ALG jest pewnym rozkładem prawdopodobieństwa $\mathcal{A}$ nad wszystkimi możliwymi algorytmami deterministycznymi,[2] tzn. zamiast pisać $\mathbf{E}[\textsc{Alg}(\sigma)]$ będziemy pisać $\mathbf{E}_{\textsc{Det}\sim\mathcal{A}}[\textsc{Det}(\sigma)]$. Popatrzmy na jego średni koszt na sekwencjach z π.

$$\begin{aligned} \mathbf{E}_{\sigma\sim\pi}\mathbf{E}_{\textsc{Det}\sim\mathcal{A}}[\textsc{Det}(\sigma)] &= \mathbf{E}_{\textsc{Det}\sim\mathcal{A}}\mathbf{E}_{\sigma\sim\pi}[\textsc{Det}(\sigma)] \\ &\geq \mathbf{E}_{\textsc{Det}\sim\mathcal{A}}[\mathcal{R} \cdot \mathbf{E}_{\sigma\sim\pi}[\textsc{Opt}(\sigma)]] \\ &= \mathbf{E}_{\sigma\sim\pi}[R \cdot \textsc{Opt}(\sigma)]], \end{aligned}$$

czyli

$$\mathbf{E}_{\sigma\sim\pi}\left(\mathbf{E}_{\textsc{Det}\sim\mathcal{A}}[\textsc{Det}(\sigma)] - R \cdot \textsc{Opt}(\sigma)\right) \geq 0$$

Zatem istnieje σ^* taka że

$$\mathbf{E}_{\textsc{Det}\sim\mathcal{A}}[\textsc{Det}(\sigma^*)] - R \cdot \textsc{Opt}(\sigma^*) \geq 0$$

czyli

$$\mathbf{E}_{\textsc{Det}\sim\mathcal{A}}[\textsc{Det}(\sigma^*)] \geq R \cdot \textsc{Opt}(\sigma^*)$$

co kończy dowód. $\blacksquare$

5.2. Wariant dla zwykłej konkurencyjności

Tę wersję pozostawiamy bez dowodu (jest w komentarzu).

Lemat 5.3 (Zasada minimaksowa Yao)**.** *Rozważmy dowolny problem minimalizacyjny. Załóżmy, że istnieje ciąg rozkładów prawdopodobieństwa $\pi_1, \pi_2, \pi_3, \ldots$ nad zbiorem wszystkich możliwych sekwencji wejściowych $\mathcal{I}$, taki że dla dowolnego algorytmu deterministycznego* DET *zachodzi*

(i) $\lim_{t\to\infty} \mathbf{E}_{\sigma\in\pi_t}[DET(\sigma)] = \infty$ *oraz*

(ii) $\liminf_{t\to\infty} \mathbf{E}_{\sigma\in\pi_t}[DET(\sigma)]/\mathbf{E}_{\sigma\in\pi_t}[OPT(\sigma)] = \mathcal{R}$.

Wtedy $\mathcal{R}$ jest dolnym ograniczeniem na konkurencyjność dowolnego zrandomizowanego algorytmu.

[2] Jest to prawda tylko dla algorytmów, których pamięć jest nieograniczona

5.3. Dolna granica na pamięć podręczną

Idea. Ustalmy dowolne M będące długością generowanego ciagu. Wykorzystamy tylko $k+1$ pierwszych stron pamięci RAM a sekwencja wejściowa będzie utworzona w następujący losowy sposób. Pierwszą stronę wejścia σ wybieramy losowo (jednostajnie) ze zbioru wszystkich $k+1$ stron, $\{p_1, p_2, \ldots, p_{k+1}\}$. Odwołanie do strony $i > 1$ jest wybierane losowo (jednostajnie) ze zbioru $\{p_1, p_2, \ldots, p_{k+1}\} \setminus I_{i-1}$, gdzie I_{i-1} jest stroną która została wybrana w kroku $i-1$. Tak powstały rozkład prawdopodobieństwa nad możliwymi wejściami oznaczamy $\pi(M)$.

Ustalmy dowolny algorytm deterministyczny DET i obliczmy, ile wynosi $\mathbf{E}_{\pi(M)}[\textsc{Det}(\sigma)]$. Odwołanie do pierwszej strony powoduje chybienie z prawdopodobieństwem $\frac{1}{k+1}$. Weźmy odwołanie do strony w kroku $i > 1$. W tym kroku algorytm ma w cache na pewno stronę I_{i-1} oraz $k-1$ innych stron. Strona I_i zostaje wybrana losowo ze zbioru $\{p_1, p_2, \ldots, p_{k+1}\} \setminus I_{i-1}$, a zatem jest w cache z prawdopodobieństwem $\frac{k-1}{k}$. Zatem oczekiwana liczba chybień i zarazem oczekiwany koszt algorytmu na losowym wejściu σ to

$$\mathbf{E}_{\pi(M)}[\textsc{Det}(\sigma)] = \frac{1}{k+1} + (M-1) \cdot \frac{1}{k} = \frac{M}{k} - \frac{1}{k \cdot (k+1)} \ .$$

A zatem $\mathbf{E}_{\pi(M)}[\textsc{Det}(\sigma)]/M \to 1/k$.

Co możemy powiedzieć o zachowaniu OPT na σ? Załóżmy przez chwilę, ze M jest nieskończone i podzielmy σ na fazy. Jaka jest oczekiwana długość takiej fazy?

Lemat 5.4. *Oczekiwana długość fazy w sekwencji σ wynosi $k \cdot H_k$.*

Dowód. Weźmy dowolną fazę. W pierwszym jej kroku wylosowana zostaje pewna strona I_0. W drugim kroku zgodnie z naszym procesem losowym wybieramy jakąś stronę $I_1 \neq I_0$. W kolejnych krokach będziemy wybierać strony aż napotkamy na stronę $I_2 \notin \{I_0, I_1\}$. Prawdopodobieństwo wylosowania takiej strony w jednym kroku to $(k-1)/k$, zatem oczekiwana liczba kroków po której to nastąpi wynosi $k/(k-1)$. Analogicznie, oczekiwana liczba kroków od momentu napotkania strony I_j do napotkania strony I_{j+1} to $k/(k-j)$. Ostatni krok (w którym odnajdujemy stronę I_k) należy już do kolejnej fazy. Zatem

$$\begin{aligned}
\mathbf{E}[\#\text{kroków w fazie}] &= 1 + 1 + \frac{k}{k-1} + \frac{k}{k-2} + \ldots + \frac{k}{1} - 1 \\
&= \frac{k}{k} + \frac{k}{k-1} + \frac{k}{k-2} + \ldots + \frac{k}{1} \\
&= k \cdot H_k \ .
\end{aligned}$$

■

Ograniczanie kosztu OPT. Jak z powyższego lematu obliczyć $\mathbf{E}_{\pi(M)}[\textsc{Opt}(\sigma)]$? Ustalmy dowolne M, to nam definiuje $\pi(M)$, wylosujmy σ zgodnie z $\pi(M)$. Niech Y_i będzie zmienną losową oznaczającą długość fazy i. Oczywiście wszystkie te zmienne mają identyczny rozkład[3], a powyżej pokazaliśmy, że $\mathbf{E}[Y_i] = k \cdot H_k$.

Pokażemy teraz ograniczenie na koszt algorytmu optymalnego. Zdefiniujemy zmienną losową $N := \max\{s : \sum_{j=1}^{s} Y_j \leq M\}$. Innymi słowy N jest liczbą faz, które jeszcze mieszczą się w sekwencji o długości M. Ponieważ koszt OPT to dokładnie 1 dla każdej rozpoczętej fazy, więc $N \leq \textsc{Opt}(\sigma) \leq N+1$. Z *podstawowego twierdzenia o procesach odnawialnych* (patrz poniżej) wynika, że

$$\lim_{M \to \infty} \frac{\mathbf{E}[N]}{M} = \frac{1}{\mathbf{E}[Y_i]} \ .$$

Twierdzenie 5.5 (Podstawowe twierdzenie procesów odnawialnych)**.** *Niech $X_1, X_2, \ldots$ będą niezależnymi zmiennymi losowymi o takim samym rozkładzie, takimi że $\mathbf{Pr}[X_i = 0] < 1$ i niech X będzie dowolną z nich. Niech $S_n = \sum_{i=1}^{n} X_i$. Niech $N(t) = \max\{n : S_n \leq t\}$ Wtedy zachodzi*

$$\lim_{t \to \infty} \frac{\mathbf{E}[N(t)]}{t} = \frac{1}{\mathbf{E}[X]} \ .$$

Przykład, że powyższe twierdzenie nie zachodzi bez lim: Y jest równe 1 z ppb. 9/10 i 100 z ppb. 1/10. Weźmy $M = 11$. Wtedy $E[Y] \approx 11$ ale $E[N] > (9/10)^{11} \cdot 11 \approx 3.45$.

Wniosek. Zatem $\lim_{M \to \infty} \mathbf{E}_{\pi(M)}[\textsc{Opt}(\sigma)] = \infty$ oraz $\lim_{M \to \infty} \mathbf{E}_{\pi(M)}[\textsc{Det}(\sigma)]/\mathbf{E}_{\pi(M)}[\textsc{Opt}(\sigma)] = H_k$. Stąd na mocy zasady minimaksowej wynika, ze H_k jest dolnym ograniczeniem na konkurencyjność dowolnego algorytmu randomizowanego.

[3] Zakładamy tutaj, że tak naprawdę sekwencja wejściowa jest nieskończona, wtedy żadna z faz nie jest ucięta, choć koncentrujemy się na początku tej sekwencji o długości M

6. Metoda podwajania

W.6A

ONLINE BIDDING. Adwersarz wybiera pewną cenę przedmiotu $u \geq 1$. Algorytm podaje ciąg $d_1, d_2, d_3, \ldots$. Gra kończy się na pierwszym elemencie o wartości co najmniej u. Koszt to suma elementów wypisanych do tej pory.

Bez straty ogólności algorytm online jest równoważny z nieskończonym monotonicznie rosnącym ciągiem liczb $d_1 < d_2 < d_3 \ldots$. Rozwiązanie OPT to po prostu ciąg równy tożsamościowo u.

Niech i będzie takie że $d_i < u \leq d_{i+1}$. Wtedy

$$\frac{\text{ALG}}{\text{OPT}} = \frac{\sum_{j=1}^{i+1} d_j}{u} \leq \frac{\sum_{j=1}^{i+1} d_j}{d_i} \ .$$

Jeśli $u < d_1$ to

$$\frac{\text{ALG}}{\text{OPT}} = \frac{d_1}{u} \leq d_1 \ .$$

A zatem współczynnik ścisłej konkurencyjności algorytmu to co najwyżej $\max\{d_1, \max_{i\geq 1}(\sum_{j=1}^{i+1} d_j)/d_i\}$. Z drugiej strony współczynnik ten nie jest lepszy, bo adwersarz może wybierać $u = 1$ bądź $u = d_i + \varepsilon$ dla dowolnie małego ε. Stąd

$$CR(\text{ALG}) = \max\left\{d_1, \max_{i\geq 1} \frac{\sum_{j=1}^{i+1} d_j}{d_i}\right\} \ .$$

Zauważmy, że biorąc $d_i = 2^{i-1}$ mamy $CR(\text{ALG}) \leq (1 + 2 + \ldots + 2^i)/2^{i-1} \leq 4$. Można też pokazać, że nie da się skonstruować algorytmu o niższym współczynniku (ćwiczenie).

6.1. Bidding vs CowPath

Przypomnijmy problem poszukiwania wejścia na pastwisko. Tam ALG również odpowiada ciągowi $(d_i)_{i=1}^{\infty}$. Jeśli adwersarz wybierze $u < d_1$ to

$$\frac{\text{ALG}}{\text{OPT}} = \frac{2 \cdot d_1 + u}{u} \leq 2 \cdot d_1 + 1 \ .$$

Natomiast jeśli istnieje i takie, że $d_i < u \leq d_{i+1}$, to

$$\frac{\text{ALG}}{\text{OPT}} = \frac{(\sum_{j=1}^{i+1} 2 \cdot d_j) + u}{u} \leq 2\frac{\sum_{j=1}^{i+1} d_j}{d_i} + 1 \ .$$

A zatem współczynnik ścisłej konkurencyjności algorytmu to co najwyżej $1 + 2 \cdot \max\{d_1, \max_{i\geq 1}(\sum_{j=1}^{i+1} d_j)/d_i\}$. Podobnie jak poprzednio współczynnika tego nie można zmniejszyć. Otrzymujemy zatem równoważność między problemami,

$$CR(\text{CowPath}) = 2 \cdot CR(\text{Bidding}) + 1 \ .$$

W szczególności implikuje to, że optymalnym współczynnikiem dla problemu CowPath jest 9.

6.2. Szeregowanie na identycznych maszynach

Dany jest ciąg zadań $a_1, a_2, \ldots$; $p_t^i = a_t$ jest czasem wykonania zadania a_t na maszynie i. Niech $L_t(i)$ będzie obciążeniem maszyny i po kroku t.

Twierdzenie 6.1. *Algorytm zachłanny jest ściśle 2-konkurencyjny.*

Dowód. Ustalmy input o T krokach. Niech j bedzie najbardziej obciążoną maszyną (na końcu). Rozważmy krok t w którym dodajemy ostatnie zadanie do tej maszyny. A zatem

$$\begin{aligned}
\text{ALG} &= L_{t-1}(j) + a_t \\
&\leq \frac{1}{m} \cdot \sum_{i=1}^{m} L_{t-1}(i) + a_t \\
&\leq \frac{1}{m} \cdot \sum_{i=1}^{m} L_T(i) + a_t \\
&\leq \text{OPT} + \text{OPT}
\end{aligned}$$

■

6.3. Szeregowanie na powiązanych maszynach

W.6B

Mamy ciąg m maszyn uszeregowanych od najwolniejszej (prędkość v_1) do najszybszej (prędkość v_m). Obciążenie zadania a_t na maszynie j to $p_t = a_t/v_j$.

Algorytm SLOWFIT(λ): uruchom zadanie a_t na najwolniejszej maszynie j, tak żeby po uruchomieniu jej obciążenie $(L_{t-1}(j) + p_t(j))$ było co najwyżej 2λ. Jeśli taka maszyna nie istnieje zwróć „FAIL".

Lemat 6.2. *Jesli* OPT $\leq \lambda$, *to* SLOWFIT$(\lambda) \leq 2\lambda$.

Dowód. Załóżmy nie wprost: istnieje taki ciąg σ, że ostatnie zadanie z tego ciągu a_t powoduje „FAIL".

Maszyna j jest *przeładowana*, jeśli $L_{t-1}(j) > \lambda$. Zauważmy, że najszybszą maszyna jest przeładowana $(L_{t-1}(m) > \lambda$ (w przeciwnym przypadku, a_t dałoby się uszeregować na maszynie m, bo $p_t(m) \leq \lambda$ (bo OPT gdzieś daje to zadanie, więc nie może być ono za duże).

Zdefiniujmy niepusty zbiór Γ szybkich przeładowanych maszyn, tj. $\Gamma = \{f+1, f+2, \ldots, m\}$, taki że maszyny $j \in \Gamma$ są przeładowane, zaś maszyna f nie jest przeładowana (lub $f = 0$).

Pokażemy, że z przeładowania wynika, że SLOWFIT trzyma więcej zadań na maszynach z Γ, zaś z własności algorytmu będzie wynikać, że trzyma tam mniej zadań.

- Obserwacja: Jeśli SLOWFIT wykonuje jakieś zadanie a_ℓ na maszynie z Γ, to OPT wykonuje to zadanie też na maszynie z Γ.

  Dlaczego tak jest? Jeśli $f = 0$ to jest to trywialnie spełnione. Załóżmy, że $f \geq 1$. Zadanie a_ℓ nie zostało wykonane na maszynie f, choć jej obciążenie było co najwyżej λ (bo $L_{\ell-1}(f) \leq L_{t-1}(f) \leq \lambda$), zatem maszyna f była na to zadanie za wolna $(p_t(f) > \lambda)$. Więc OPT nie mógł wykonać zadania a_ℓ na maszynach $1, 2, \ldots, f$.

- Niech S_j i S_j^* będą indeksami zadań wykonanych przez SLOWFIT i OPT na maszynie j. Z poprzedniego punktu wynika, że $\bigcup_{j\in\Gamma} S_j \subseteq \bigcup_{j\in\Gamma} S_j^*$. Zatem:

  $$\sum_{j\in\Gamma}\sum_{\ell\in S_j} a_\ell = \sum_{j\in\Gamma} v_j \sum_{\ell\in S_j} p_\ell(j) > \sum_{j\in\Gamma} v_j \cdot \lambda.$$

  Oraz

  $$\sum_{j\in\Gamma}\sum_{\ell\in S_j^*} a_\ell = \sum_{j\in\Gamma} v_j \sum_{\ell\in S_j^*} p_\ell(j) \leq \sum_{j\in\Gamma} v_j \cdot \lambda.$$

  Ale $\sum_{j\in\Gamma}\sum_{\ell\in S_j} a_\ell \leq \sum_{j\in\Gamma}\sum_{\ell\in S_j^*}$, a więc otrzymujemy sprzeczność.

■

\+ podwajanie (spisane na kartce).

7. Programowanie liniowe: Set Cover

7.1. Przykładowe programy liniowe

Rozważmy następujący program liniowy (P):

$$\begin{aligned} \text{zminimalizuj } & P = 2x_1 + 4x_2 + 4x_3 \\ \text{z zachowaniem ograniczeń: } & x_1 + x_2 \geq 3 \\ & x_2 + x_3 \geq 1 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

Ograniczenie optymalnej wartości P od góry jest proste: wystarczy podać *jakieś* x_1, x_2 i x_3 spełniające ograniczenia. Przykładowo dla $x_1 = 1$, $x_2 = 2$ i $x_3 = 0$ otrzymujemy $P = 10$.

Jak możemy wygenerować ograniczenie P od dołu? Zauważmy, że z warunków wynika, że:

$$\begin{aligned} 2x_1 + 2x_2 &\geq 6 \\ 2x_2 + 2x_3 &\geq 2 \end{aligned}$$

czyli

$$2x_1 + 4x_2 + 2x_3 \geq 8$$

A zatem

$$\begin{aligned} P &= 2x_1 + 4x_2 + 4x_3 \\ &\geq 2x_1 + 4x_2 + 2x_3 \\ &\geq 8 \end{aligned}$$

Co się tu wydarzyło? Pomnożyliśmy wymagane ograniczenia przez jakieś *nieujemne* współczynniki y_1 i y_2 (w naszym przypadku wzięliśmy $y_1 = 2$ i $y_2 = 2$) i następnie zsumowaliśmy otrzymane nierówności. Otrzymaliśmy, że

$$y_1 \cdot (x_1 + x_2) + y_2 \cdot (x_2 + x_3) \geq 3y_1 + 1y_2$$

czyli

$$y_1 \cdot x_1 + (y_1 + y_2) \cdot x_2 + y_2 \cdot x_3 \geq 3y_1 + 1y_2$$

Kiedy taka nierówność jest dla nas przydatna do ograniczania P od dołu? Potrzebujemy, żeby współczynniki przy x_i w wyrażeniu P jak i w powyższej nierówności były w odpowiedniej relacji, tj.

$$\begin{aligned} y_1 &\leq 2 \\ y_1 + y_2 &\leq 4 \\ y_2 &\leq 4 \end{aligned}$$

Jeśli wybierzemy y_j spełniające te nierówności, to wyrażenie $D = 3y_1 + 1y_2$ będzie dolnym ograniczeniem na P. Trochę bardziej formalnie możemy myśleć o D jako o funkcji y_1, y_2 zaś o P jako o funkcji x_1, x_2, x_3. Wtedy otrzymujemy, że dla odpowiednio ograniczonych wartości y_j i x_i mamy:

$$D(y_1, y_2) \leq P(x_1, x_2, x_3)$$

To jest słaba wersja dualności, a D nazywamy programem dualnym do danego.

7.2. Online Set Cover

ONLINE SET COVER. 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 $S \in \mathcal{F}$, $c_S \geq 1$ 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 $\sum_{S \ni e_i} x_S \geq 1$. 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.

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

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 $\sum_{S \ni e_i} x_S \geq 1$ zmienną y_i.

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

Zauważmy, że wartość optymalnego rozwiązania dla (D_0) jest również równa zeru. Dodatkowo ze słabej własności dualności (która wprost wynika ze sposobu w jaki utworzyliśmy program (D_k) wynika, że $\textsc{Opt}(D_k) \leq \textsc{Opt}(P_k)$.

Będziemy teraz konstruować ciąg rozwiązań $\textsc{Alg}(P_i)$ i $\textsc{Alg}(D_i)$ układów (P_i) i (D_i) taki, że prawdziwe są następujące kluczowe warunki.

W1. Rozwiązanie $\textsc{Alg}(P_i)$ spełnia wszystkie ograniczenia (P_i) i dodatkowo jest monotoniczne.

W2. Rozwiązanie $\textsc{Alg}(D_i)$ spełnia wszystkie ograniczenia (D_i).

W3. $\Delta\textsc{Alg}(P_i) \leq R \cdot \Delta\textsc{Alg}(D_i)$.

W powyższych relacjach $R = 2 \cdot \log(2m + 2) = O(\log m)$.

Lemat 7.1. *Jeśli algorytm generuje rozwiązania układów (P_i) i (D_i) zgodne z warunkami W1, W2 oraz W3, to jego jest ściśle R-konkurencyjny.*

Dowód. Wystarczy pokazać, że dla każdego kroku k zachodzi $\textsc{Alg}(P_k) \leq R \cdot \textsc{Opt}(P_k)$. Sumując warunek W3 po pierwszych k krokach orzymujemy

$$\textsc{Alg}(P_k) = \sum_{i=1}^{k} \Delta\textsc{Alg}(P_i) \leq \sum_{i=1}^{k} R \cdot \Delta\textsc{Alg}(D_i) = R \cdot \textsc{Alg}(D_k) \ ,$$

a stąd

$$\textsc{Alg}(P_k) \leq R \cdot \textsc{Alg}(D_k) \leq R \cdot \textsc{Opt}(D_k) \leq R \cdot \textsc{Opt}(P_k) \ .$$

Środkowa nierówność wynika z warunku W2 (dopuszczalności rozwiązania $\textsc{Alg}(D_k)$). ■

Algorytm dla kroku k. W programie (P_k) pojawia się nowy warunek $\sum_{S \ni e_k} x_S \geq 1$ 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 $\sum_{e_i \in S} y_i \leq c_S$, 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 zbioru S zawierającego e_k wykonujemy $x_S \leftarrow (1 + 1/c_S) \cdot x_S + 1/(m \cdot c_S)$.

2. $y_k \leftarrow y_k + 1/\log(2m + 2)$

Zauważmy jednak już teraz, że żeby rozwiązanie (D_k) było dopuszczalne, liczba o którą zwiększamy y_k powinna być mała i mała powinna być liczba iteracji, a zatem zwiększanie x_S powinno być jak najszybsze. Z drugiej strony, żeby minimalizować współczynnik konkurencyjności zależy nam na dokładnie odwrotnych zależnościach.

Lemat 7.2. *Powyższy algorytm spełnia warunek W1.*

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 $1/(m \cdot c_S)$, po skończonej liczbie iteracji otrzymamy $\sum_{S \ni e_k} x_S \geq 1$. ■

Lemat 7.3. *Powyższy algorytm spełnia warunek W3.*

Dowód. Dla warunku W3 zauważmy, że w każdej iteracji pętli $\Delta\textsc{Alg}(D_k) = 1/\log(2m + 2)$, natomiast na zwiększenie wartości rozwiązania układu (P_k) wpływa zwiększenie wartości x_S dla zbiorów S zawierających e_k. Zatem

$$\Delta\textsc{Alg}(P_k) = \sum_{S \ni e_k} c_S \cdot \Delta x_S = \sum_{S \ni e_k} c_S \cdot \left(\frac{1}{c_S} \cdot x_S + \frac{1}{m \cdot c_S} \right) = \sum_{S \ni e_k} x_S + \sum_{S \ni e_k} \frac{1}{m} < 1 + 1 = 2 \ .$$

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. ■

Lemat 7.4. *Powyższy algorytm spełnia warunek W2.*

Dowód. Zobaczmy najpierw, dla dowolnego zbioru S, ile razy w ciągu działania algorytmu może wykonać się przypisanie $x_S \leftarrow (1 + 1/c_S) \cdot x_S + 1/(m \cdot c_S)$. Nazwijmy takie przypisanie „zwiększaniem x_S" a wartość x_S po ℓ zwiększeniach oznaczmy przez $x_S^{(\ell)}$. Łatwo zauważyć, że

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

Niech t_S oznacza liczbę zwiększeń x_S w wykonaniu algorytmu. Zauważmy, że $x_S^{t_S - 1} < 1$ (żeby ostatnie zwiększenie miało szanse się wykonać). Możemy zatem wyznaczyć górne ograniczenie na t_S.

$$t_S < \frac{\log(m+1)}{\log(1 + 1/c_S)} + 1 \leq \log(m+1) \cdot c_S + 1 \leq \log(2m+2) \cdot c_S \ .$$

W ograniczeniach wykorzystaliśmy, że $c_S \geq 1$ (co implikuje $\log(1 + 1/c_S) \geq 1/c_S$).

Niech k będzie całkowitą liczbą elementów do pokrycia w sekwencji wejściowej. Żeby pokazać warunek W2 wystarczy pokazać, że na końcu działania algorytmu wszystkie nierówności w (D_k) są spełnione. Weźmy dowolną taką nierówność $\sum_{e_i \in S} y_i \leq c_S$ (dla dowolnego $S \in \mathcal{F}$).

W momencie kiedy algorytm zwiększa dowolne y_i wchodzące w skład lewej strony tej nierówności (zauważmy, że jest wtedy krok i) wykonywana jest operacja zwiększenia dla wszystkich zbiorów S zawierających e_i. W szczególności zwiększany jest wtedy x_S. Jak pokazaliśmy wyżej liczba takich zwiększeń wynosi co najwyżej $\log(2m + 2) \cdot c_S$, a zatem całkowity wartość lewej strony nierówności (na końcu działania algorytmu) wynosci co najwyżej $\frac{1}{\log(2m+2)} \cdot \log(2m + 2) \cdot c_S = c_S$. ■

8. Routing

W tym rozdziale zajmiemy się podproblemem routingu, tj. wyborem tras. Dla uwagi ustalmy pewien graf $G = (V, E)$. Każda krawędź ma przepustowość U. Wejście dla problemu routingu składa się z par (s_j, t_j), a dla każdej takiej pary algorytm musi zdecydować czy połączenie odrzucić czy zaakceptować i w tym drugim przypadku wybrać w grafie G jakąś ścieżkę P_j łączącą s_j z t_j. (To jest tzw. virtual circuit routing). Celem jest maksymalizacja liczby zaakceptowanych połączeń.

Dla dowolnej krawędzi e definiujemy $\ell_j(e)$, *obciążenie krawędzi* po kroku j, jako liczbę przechodzących przez nią scieżek. W generowanym przez algorytm rozwiązanie musi zachodzić $\ell_j(e) \leq U$.

8.1. Małe przepustowości

N-linią nazywamy graf z N wierzchołkami numerowanych od 0 do $N-1$ połączonych w linię, gdzie $U = 1$.

Twierdzenie 8.1. *Rozważmy N-linię. Wtedy każdy deterministyczny algorytm online dla dopuszczania połączeń ma współczynnik ścisłej 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. ■

8.2. Przypadek dużych przepustowości

Będziemy zakładać, że U jest odpowiednio duże (potem się okaże że musi być co najmniej $\ln(m+1)$, gdzie m jest liczbą krawędzi.)

Intuicja. Wybór ścieżki powinien optymalizować dwie rzeczy:

- ścieżka powinna być jak najkrótsza
- ścieżka powinna omijać już mocno obciążone krawędzie

Jaki wybór optymalizuje pkt. pierwszy a jaki drugi? Pomiędzy tymi wyborami można popatrzeć na wartości $w_e := 2^{\ell(e)}$ dla krawędzi e i wybierać ścieżkę, która ma jak najmniejszą sumaryczną wagę. Dodatkowo, jeśli najmniejsza sumaryczna waga jest powyżej pewnego progu, to nie przyjmujemy tego połączenia.

Uwaga: minimalizacja $\sum_{e\in P} w_e$ minimalizuje też $Q_P = \log \sum_{e\in P} w_e$. Można zauważyć, że

$$\log(\max_{e\in P} w_e) \leq Q_P \leq \log(m \cdot \max_{e\in P} w_e) = \log m + \log(\max_{e\in P} w_e)$$

czyli

$$\max_{e\in P} \ell(e) \leq Q_P \leq \log m + \max_{e\in P} \ell(e)$$

Oznacza to że minimalizujemy wielkość, która z dokładnością do addytywnego składnika $\log m$ jest tak samo duża jak $\max_{e\in P} \ell(e)$ (load na najbardziej obciążonej krawędzi).

Program liniowy. Rozważmy (całkowitoliczbowy) program liniowy opisujący problem routingu online. Dla żądania $r_i = (s_i, t_i)$ przedstawionego w kroku i niech $\mathbb{P}(r_i)$ oznacza zbiór wszystkich możliwych ścieżek od s_i do t_i. Dla danego żądania r_i i ścieżki $P \in \mathbb{P}(r_i)$ zmienna $f(r_i, P)$ jest równa 1 jeśli dla obsługi żądania r_i wybraliśmy ścieżkę P i 0 w przeciwnym przypadku. Zatem po kroku k algorytm musi wygenerować rozwiązanie którego zyskiem jest

$$\sum_{i=1}^{k} \sum_{P\in\mathbb{P}(r_i)} f(r_i, P) \ ,$$

przy zachowaniu warunków:

$$\sum_{P\in\mathbb{P}(r_i)} f(r_i, P) \leq 1 \qquad \text{dla każdego } i \in \{1, \ldots, k\},$$

$$\sum_{i=1}^{k} \sum_{P\in\mathbb{P}(r_i): e\in P} f(r_i, P) \leq U \qquad \text{dla każdego } e \in E,$$

$$f(r_i, P) \in \{0, 1\} \qquad \text{dla każdego } i \in \{1, \ldots, k\}, P \in \mathbb{P}(r_i).$$

Program ten ma wykładniczą liczbę zmiennych, ale nie będziemy się tym przejmować. Co więcej nasz algorytm da się zaimplementować w czasie wielomianowym. Nazwijmy ten program $\mathcal{P}_k^{INT}$. Niech $\mathcal{P}_k$ będzie liniową relaksacją powyższego zagadnienia, w którym celem jest maksymalizacja zysku, a warunek $f(r_i, P) \in \{0, 1\}$ został zastąpiony przez $f(r_i, P) \geq 0$. Uwaga: $f(r_i, P) \leq 1$ jest implikowane przez $\sum_{P \in \mathbb{P}(r_i)} f(r_i, P) \leq 1$ a zatem zbędne.

Programem dualnym do $\mathcal{P}_k$ jest następujące zagadnienie minimalizacyjne $\mathcal{D}_k$:

$$\begin{aligned}
\text{minimalizuj: } & \sum_{i=1}^{k} z_i + U \cdot \sum_{e \in E} x_e & & \\
\text{przy warunkach: } & z_i + \sum_{e \in P} x_e \geq 1 & & \text{dla każdych } i \in \{1, \ldots, k\},\ P \in \mathbb{P}(r_i), \\
& z_i \geq 0 & & \text{dla każdego } i \in \{1, \ldots, k\}, \\
& x_e \geq 0 & & \text{dla każdego } e \in E.
\end{aligned}$$

Zauważmy, że ze słabej dualności i z tego, że $\mathcal{P}_k$ jest relaksacją $\mathcal{P}_k^{\text{INT}}$ wynika, że

$$\textsc{Opt}(\mathcal{P}_k^{\text{INT}}) \leq \textsc{Opt}(\mathcal{P}_k) \leq \textsc{Opt}(\mathcal{D}_k).$$

Będziemy generować parę rozwiązań $\textsc{Alg}(\mathcal{P}_k^{\text{INT}})$ i $\textsc{Alg}(\mathcal{D}_k)$. Oczywiście mamy wtedy

$$\textsc{Alg}(\mathcal{P}_k^{\text{INT}}) \leq \textsc{Opt}(\mathcal{P}_k^{\text{INT}}) \leq \textsc{Opt}(\mathcal{P}_k) \leq \textsc{Opt}(\mathcal{D}_k) \leq \textsc{Alg}(\mathcal{D}_k).$$

Wystarczy zatem, że odpowiednio związemy ze sobą $\textsc{Alg}(\mathcal{P}_k^{\text{INT}})$ i $\textsc{Alg}(\mathcal{D}_k)$.

Algorytm. $A = 2\ln(m+1)/U$. Algorytm jest parametryzowany przez A i T, które zdefiniujemy potem. W kroku k algorytm ustawia $z_k = 0$ i sprawdza, czy istnieje ścieżka $P \in \mathbb{P}(r_k)$, taka że $\sum_{e \in P} x_e < 1$. (Można wybrać taką minimalizującą sumę x_e, ale nie trzeba). Jeśli tak, to

- zaakceptuj żądanie i zwróć P jako odpowiedź,
- dla każdej krawędzi $e \in P$ przypisz $x_e \leftarrow x_e \cdot (1 + A) + A/m$.
- przypisz $z_k \leftarrow 1$ (można oszczędniej, ale po co)

Jeśli taka ścieżka P nie istnieje, to odrzuć żądanie r_k.

Własności.

Lemat 8.2. *W każdym momencie zachodzi*

$$x_e = \frac{(1 + A)^{\ell(e)} - 1}{m}.$$

Dowód. Dowód indukcyjny. Prawda na początku, bo $\ell(e) = 0 = x_e$. Kiedy wybieramy ścieżkę P i inkrementujemy $\ell(e)$ dla każdego $e \in P$, to wykonujemy też podstawienie $x_e \leftarrow x_e \cdot (1 + A) + A/m$. Zatem

$$\begin{aligned}
x'_e &= x_e \cdot (1 + A) + A/m \\
&= \frac{(1 + A)^{\ell(e)} - 1}{m} \cdot (1 + A) + \frac{A}{m} \\
&= \frac{(1 + A)^{\ell(e)+1} - 1 - A}{m} + \frac{A}{m} \\
&= \frac{(1 + A)^{\ell'(e)} - 1}{m}
\end{aligned}$$

■

Lemat 8.3. *$\textsc{Alg}(\mathcal{D}_k)$ jest dopuszczalnym rozwiązaniem.*

Dowód. W kroku k nowymi warunkami w $\mathcal{D}_k$ są nierówności $z_k + \sum_{e \in P} x_e$ (dla każdej ścieżki $P \in \mathbb{P}(r_k)$). Jeśli żądanie jest przyjęte, to nierówności te spełniają się przez ustawienie $z_k \leftarrow 1$. Jeśli jest odrzucone, to znaczy, że dla wszystkich ścieżek $P \in \mathbb{P}(r_k)$ zachodzi $\sum_{e \in P} x_e \geq 1$, więc nierówności są spełnione. ■

Lemat 8.4. *W kroku k, zachodzi $\Delta\textsc{Alg}(\mathcal{D}_k) \leq (1 + 2 \cdot U \cdot A) \cdot \Delta\textsc{Alg}(\mathcal{P}_k)$.*

Dowód. Jeśli żądanie nie jest akceptowane, to $\Delta\text{ALG}(\mathcal{D}_k) = \Delta\text{ALG}(\mathcal{P}_k) = 0$. Jeśli żądanie jest akceptowane, to $\Delta\text{ALG}(\mathcal{P}_k) = 1$, natomiast

$$\Delta\text{ALG}(\mathcal{D}_k) = z_k + U \cdot \sum_{e \in P} \Delta x_e = 1 + U \cdot (\sum_{e \in P} A \cdot x_e + \sum_{e \in P} A/m) = 1 + U \cdot A + U \cdot A. \qquad \blacksquare$$

Lemat 8.5. *ALG($\mathcal{P}_k$) jest dopuszczalnym rozwiązaniem jeśli $U \geq \ln(m+1)$ i $A = (2\ln(m+1))/U$.*

Dowód. Zobaczmy, co się dzieje, w momencie kiedy dokładamy nową ścieżkę P. Żeby tak się stało, musiało zachodzić $\sum_{e \in P} x_e < 1$, czyli:

$$\frac{(1+A)^{\ell(e)} - 1}{m} < 1.$$

Stąd

$$\ell(e) < \frac{\ln(m+1)}{\ln(1+A)} \leq \frac{2}{A} \cdot \ln(m+1) \leq U$$

Przedostatnia nierówność jest prawdziwa jeśli $A \in [0, 2]$ (wtedy $\ln(1+A) \geq A/2$). Ostatnia nierówność wymaga, żeby $A \geq (2\ln(m+1))/U$. $\blacksquare$

Zatem jeśli $U \geq \ln(m+1)$, to wybierając $A = (2\ln(m+1))/U$ gwarantujemy, że rozwiązanie algorytmu jest dopuszczalne i dodatkowo mamy $\text{ALG}(\mathcal{D}_k) \leq (1 + 2 \cdot U \cdot A) \cdot \text{ALG}(\mathcal{P}_k) = (1 + 4\ln(m+1)) \cdot \text{ALG}(\mathcal{P}_k^{\text{INT}})$

8.3. TEGO NIE ZDAZYLEM: Małe przepustowości: randomizacja na linii

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.[4]

> 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 8.6. *Algorytm CRS jest $\lceil \log N \rceil$-konkurencyjny.*

Dowód. Dla uproszczenia założymy, że $N = 2^k$. 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

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

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

$$\mathbf{E}[\text{CRS}(\sigma)] \geq \sum_{i=1}^{\log N} \frac{1}{\log N} \cdot o_i = \frac{1}{\log N} \cdot \text{OPT}(\sigma) \ .$$

Można rozszerzyć dowód na dowolne N, niekoniecznie będące potęgą dwójki, otrzymując tezę twierdzenia. $\blacksquare$

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

9. Problem k serwisantów

Do zdefinowania problemu k-serwer przydatne będzie przypomnienie następującego pojęcia. Przestrzenią metryczną $(\mathcal{X}, d)$ jest zbiór punktów $\mathcal{X}$ wraz ze zdefiniowaną funkcją odległości d spełniającą dla dowolnych punktów $x, y, z \in \mathcal{X}$ następujące warunki: W.7A

(i) $d(x, y) = 0 \Leftrightarrow x = y$;

(ii) $d(x, y) = d(y, x)$;

(iii) $d(x, y) \leq d(x, z) + d(z, y)$.

k SERWISANTÓW (k-SERVER). Ustalmy pewna przestrzeń metryczną. Ciąg wejściowy to ciąg punktów $(\sigma_t)_t$ z tej przestrzeni. Algorytm dysponuje zbiorem k serwerów; każdy serwer zajmuje określony punkt z przestrzeni. W kroku t algorytm może dowolnie przemieścić swoje serwery, ale pod koniec takiego ruchu, co najmniej jeden z serwerów musi być w punkcie σ_t. Algorytm płaci za sumaryczną drogę przebytą przez swoje serwery.

Zauważmy, że dla metryki dyskretnej problem jest równoważny problemowi pamięci podręcznej.

Obserwacja 9.1. *Algorytm zachłanny nie jest konkurencyjny.*

9.1. Algorytm Double Coverage

Warto zacząć od Greedy i powiedzieć, że można go trochę usprawnić przesuwając kiedyś dalszy serwer (np. starajac się zrównoważyć koszt obu serwerów). Jak zrobi się to w leniwy sposób, to dostaje się początek DC.

W tej części przedstawimy algorytm dla przypadku, w którym przestrzeń metryczna jest prostą rzeczywistą. Algorytm DOUBLECOVERAGE (DC) w każdym kroku t wykonuje następującą operację. Jeśli σ_t leży na lewo (lub na prawo) od wszystkich serwerów algorytmu, to DC przenosi do σ_t najbliższy serwer. Jeśli natomiast σ_t leży pomiędzy dwoma serwerami s_a i s_b, to DC przesuwa je z jednakową prędkością w kierunku σ_t do momentu, kiedy jeden z nich dotrze do punktu σ_t.

Tu warto pokazać na 2 serwerach, caczego jak weźmiemy potencjał równy minimalnemu matchingowi, to nie wystarczy: w przypadku gdy żądanie jest całkowicie z lewej strony, to mamy nadmiarowy potencjał a jeśli żądanie jest pomiędzy serwerami, to brakuje nam potencjału.

W.6B

Twierdzenie 9.2. *Algorytm DC jest k-konkurencyjny*

Dowód. Niech $M_{\min}$ oznacza koszt minimalnego skojarzenia między serwerami DC i serwerami OPT. Serwery algorytmu DC oznaczamy przez $s_1, s_2, \ldots, s_k$. Niech $\sum_{\text{DC}}$ będzie sumą odległości między wszystkimi parami serwerów DC, tj. $\sum_{\text{DC}} = \sum_{i<j} d(s_i, s_j)$. Definiujemy następującą nieujemną funkcję potencjału:

$$\Phi = k \cdot M_{\min} + \sum\nolimits_{\text{DC}}$$

Rozważmy krok t. Podzielimy ten krok na dwie części. W pierwszej swoje serwery przesunie OPT, w drugiej algorytm DC. Dla każdej części z osobna udowodnimy, że zachodzi nierówność

$$\text{DC} + \Delta\Phi \leq k \cdot \text{OPT} \ . \tag{7}$$

Dowód dla pierwszej części jest łatwy. Algorytm DC nie ponosi żadnego kosztu a część potencjału, $\sum_{\text{DC}}$, nie zmienia się. Jeśli OPT przesuwa jeden ze swoich serwerów o d to koszt obecnego skojarzenia zwiększa się co najwyżej o d, a zatem koszt minimalnego skojarzenia wzrasta również co najwyżej o d.

Dla dowodu drugiej częsci rozpatrzmy dwa przypadki możliwych ruchów DC. W pierwszym przypadku DC przesuwa jeden serwer s_a (leżący najbardziej na lewo lub najbardziej na prawo) w stronę punktu σ_t. Przy przesunięciu na odległość d, $\sum_{\text{DC}}$ wzrasta o $(k-1) \cdot d$. Jak wygląda minimalne skojarzenie na początku drugiej części? Serwery OPT już wykonały swój ruch, a zatem jeden z nich jest na pozycji σ_t. Można pokazać, że istnieje minimalne skojarzenie w którym serwer s_a jest skojarzony z σ_t. Stąd wynika, że przesunięcie serwera s_a do σ_t zmniejsza koszt tego skojarzenia (a więc i koszt minimalnego skojarzenia) o co najmniej d. Stąd całkowita zmiana potencjału to $\Delta\Phi \leq -k \cdot d + (k-1) \cdot d = -d$. Zatem koszt przesunięcia serwera jest amortyzowany przez spadek potencjału.

W drugim przypadku σ_t leży między dwoma serwerami s_a i s_b. Załóżmy, że każdy z nich przesunął się o d. Na początku drugiej części istnieje minimalne skojarzenie, w którym jeden z nich jest skojarzony z σ_t (ćwiczenie). Załóżmy, że jest to serwer s_a. Wtedy na skutek ruchu serwera s_a wartość $M_{\min}$ maleje o d. Z drugiej strony (być może) b oddala się swojego skojarzonego serwera, powodując wzrost $M_{\min}$ o co najwyżej d. Zatem $\Delta M_{\min} \leq 0$. Jaka jest natomiast zmiana w części $\sum_{\mathrm{DC}}$ potencjału? Zmieniają się tylko odległości między parami, w których jednym z elementów jest s_a lub s_b. Niech A oznacza zbiór serwerów na lewo od s_a i s_b, a B zbiór serwerów na prawo od nich. Wtedy suma $\sum_{s\in A} d(s, s_a)$ rośnie o $|A| \cdot d$ a suma $\sum_{s\in A} d(s, s_b)$ maleje o $|A| \cdot d$. Podobnie jest ze z odległościami między s_a, s_b i zbiorem B. Na zmianę $\sum_{\mathrm{DC}}$ wpływa zatem jedynie zmniejszająca się odległość między s_a i s_b — zmiana ta wynosi $2d$. Stąd $\Delta\Phi \leq -2d$, co kończy dowód (7). ■

Tu można wspomnieć, że dla drzew jest praktycznie tak samo.

9.2. Dolne ograniczenie

Obserwacja 9.3. *Bez straty ogólności możemy założyć, że algorytm jest leniwy, tj. po żądaniu przesuwa co najwyżej jeden serwer (do miejsca żądania).*

Lemat 9.4. *Dla dowolnej przestrzeni metrycznej, która ma więcej niż k punktów, współczynnik konkurencyjnosci dowolnego algorytmu deterministycznego* Alg *dla problemu k-serwer wynosi co najmniej k.*

Dowód. Weźmy dowolny algorytm deterministyczny Alg. Bez straty ogólności, możemy założyć, że jest on algorytmem leniwym, tj. w jednym kroku t przemieszcza tylko jeden serwer. Niech $v_1, v_2, \ldots, v_k$ będą punktami przestrzeni, w których początkowo znajdują się serwery. Niech v_{k+1} będzie dowolnym innym punktem. Zbiór $\{v_1, v_2 \ldots, v_{k+1}\}$ oznaczamy przez S; będziemy używać tylko tych wierzchołków. Na początku Alg nie ma serwera w v_{k+1}; mówimy, że Alg ma *dziurę* w v_{k+1}. Sekwencję wejściową σ generujemy w tradycyjny sposób: żądanie jest generowane zawsze w tym punkcie ze zbioru S, w którym Alg ma dziurę.

Weźmy zbiór k algorytmów $A_1, A_2, \ldots, A_k$. W pierwszym kroku A_i zmienia swoją konfigurację na taką, że ma dziurę w wierzchołku v_i. Każdy z nich płaci za to pewną stałą zależną tylko od przestrzeni metrycznej, oznaczmy górne ograniczenie na wszystkie te stałe przez α. Podzielmy teraz koncepcyjnie każdy krok t na cztery etapy:

1. Występuje żądanie w punkcie σ_t.
2. Algorytm Alg przesuwa swój (jeden) serwer do σ_t.
3. Wszystkie algorytmy $(\mathrm{Alg}, A_1, \ldots, A_k)$ obsługują żadanie.
4. Algorytmy $A_1, \ldots, A_k$ przesuwają swoje serwery.

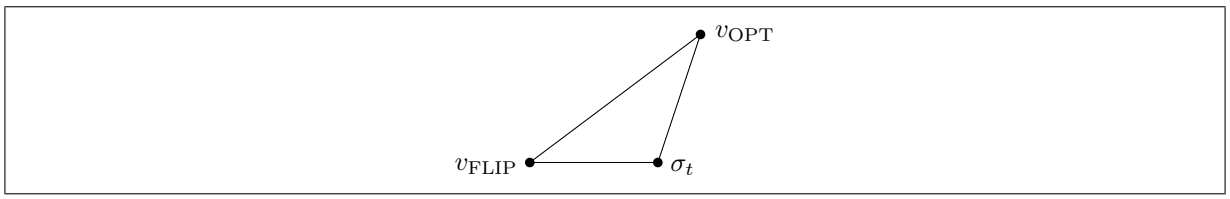
Działanie algorytmów A_i określimy definiując następujący niezmiennik. Chcemy, żeby na końcu kroku zbiór dziur algorytmów $\mathrm{Alg}, A_1, A_2, \ldots, A_k$ był równy S.

Powyższy niezmiennik jest spełniony w pierwszym kroku; pokażemy jak go zachować w kolejnych. Załóżmy, że odwołanie występuje w wierzchołku v_ℓ i Alg przesuwa tam stronę z wierzchołka v_k. Istnieje dokładnie jeden algorytm (nazwijmy go A_m), który ma dziurę w wierzchołku v_k. W następnym kroku A_m przesuwa serwer z wierzchołka v_ℓ do v_k (czyli w dokładnie przeciwną stronę niż algorytm Alg). Po takim ruchu niezmiennik nadal zachodzi, bo Alg i A_m zamienią się dziurami. Zauważmy, że A_m nie mógłby przenieść serwera od razu w tym samym kroku, gdyż wtedy nie odpowiadałby poprawnie na żądania sekwencji wejściowej.

Dodatkowo z symetryczności metryki wynika, że sumaryczny koszt wszystkich algorytmów A_i w tym kroku jest taki sam jak koszt algorytmu Alg. Dla całej sekwencji σ dostajemy zatem

$$\mathrm{Alg}(\sigma) = \sum_{i=1}^{k} (A_i(\sigma) - \alpha) \ .$$

Dlatego też jeden z algorytmów A_i (nazwijmy go A) ma koszt co najwyżej $1/k \cdot \mathrm{Alg}(\sigma) + \alpha$. Stąd $\mathrm{Opt}(\sigma) \leq A(\sigma) \leq 1/k \cdot \mathrm{Alg}(\sigma) + \alpha$. Biorąc odpowiednio kosztowną sekwencję σ, składnik α staje się zaniedbywalny i otrzymujemy tezę lematu. ■



Rysunek 1: Ilustracja dla etapu 1 algorytmu FLIP

10. Przenoszenie pliku

10.1. Algorytm randomizowany

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 $D > 1$.

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.

Rozważmy następujący algorytm FLIP. Algorytm ten w kroku t obsługuje żądanie z wierzchołka σ_t i przenosi plik do σ_t z prawdopodobieństwem $\frac{1}{2 \cdot D}$.

Twierdzenie 10.1. *Algorytm FLIP jest 3-konkurencyjny przeciwko adwersarzowi adaptującemu się.*

Dowód. Zdefiniujmy następującą funkcję potencjału

$$\Phi = 3 \cdot D \cdot d(v_{\text{FLIP}}, v_{\text{OPT}}) \ ,$$

gdzie v_{FLIP} i v_{OPT} są wierzchołkami, które przechowują pliki algorytmów FLIP i OPT. Wybierzmy dowolną sekwencję wejściową σ.

Funkcja Φ jest nieujemna, a jeśli FLIP i OPT zaczynają ze plikami w tym samym wierzchołku, jest równa początkowo zero. Wystarczy zatem pokazać, że dla dowolnego kroku t zachodzi:

$$\mathbf{E}[\text{FLIP}(t)] + \mathbf{E}[\Delta\Phi(t)] \leq 3 \cdot \text{OPT}(t) \ . \tag{8}$$

Podobnie jak w poprzednich dowodach, dzielimy krok t na dwa etapy.

1. Algorytmy FLIP i OPT płacą za odwołanie do pliku; algorytm FLIP (opcjonalnie) przenosi plik do wierzchołka v'_{FLIP}.
2. Algorytm OPT (opcjonalnie) przenosi swój plik do wierzchołka v'_{OPT}.

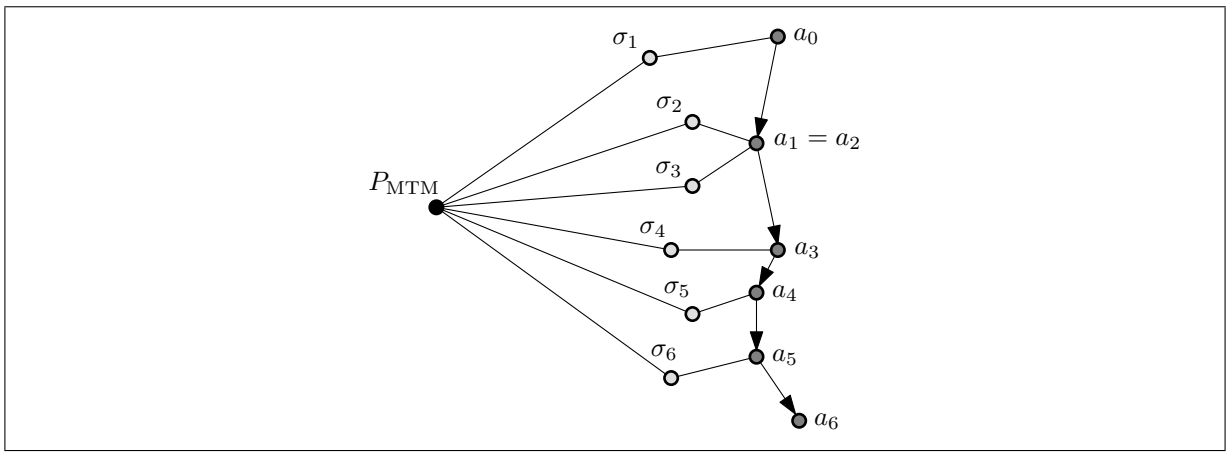
Pokażemy, że (8) zachodzi dla każdego etapu z osobna.

Etap 1. Do pliku odwołuje się komputer σ_t. Sytuacja jest zaprezentowana na poniższym rysunku. Mamy:

$$\begin{aligned} \text{OPT} &= d(v_{\text{OPT}}, \sigma_t) \\ \mathbf{E}[\text{FLIP}] &= d(v_{\text{FLIP}}, \sigma_t) + \frac{1}{2 \cdot D} \cdot D \cdot d(v_{\text{FLIP}}, \sigma_t) \\ &= \frac{3}{2} \cdot d(v_{\text{FLIP}}, \sigma_t) \ . \end{aligned}$$

Podobnie możemy ograniczyć zmianę potencjału. Z prawdopodobieństwem $1 - \frac{1}{2 \cdot D}$ nie zmienia się on podczas tego etapu, a z prawdopodobieństwem $\frac{1}{2 \cdot D}$ zmienia się w związku z przeniesieniem pliku do σ_t z $3 \cdot D \cdot d(v_{\text{FLIP}}, v_{\text{OPT}})$ na $3 \cdot D \cdot d(\sigma_t, v_{\text{OPT}})$. Zatem

$$\begin{aligned} \mathbf{E}[\Delta\Phi] &= \frac{1}{2 \cdot D} \cdot 3 \cdot D \cdot [d(\sigma_t, v_{\text{OPT}}) - d(v_{\text{FLIP}}, v_{\text{OPT}})] \\ &= \frac{3}{2} \cdot [d(\sigma_t, v_{\text{OPT}}) - d(v_{\text{FLIP}}, v_{\text{OPT}})] \ . \end{aligned}$$



Rysunek 2: Ilustracja algorytmu MTM w jednej fazie

Łącząc podane wyżej równości i korzystając z warunku trójkąta otrzymujemy

$$\begin{aligned}\mathbf{E}[\text{FLIP}] + \mathbf{E}[\Delta\Phi] &= \frac{3}{2} \cdot [d(v_{\text{FLIP}}, \sigma_t) + d(\sigma_t, v_{\text{OPT}}) - d(v_{\text{FLIP}}, v_{\text{OPT}})] \\ &\leq \frac{3}{2} \cdot [d(\sigma_t, v_{\text{OPT}}) + d(\sigma_t, v_{\text{OPT}})] \\ &= 3 \cdot \text{OPT} \ .\end{aligned}$$

Zatem (8) zachodzi.

Etap 2. W tym przypadku FLIP znajduje się już w wierzchołku v'_{FLIP} (być może $v'_{\text{FLIP}} = v_{\text{FLIP}}$). Oczywiście $\text{FLIP} = 0$, $\text{OPT} = D \cdot d(v_{\text{OPT}}, v'_{\text{OPT}})$. Jednocześnie potencjał zmienia się z $3 \cdot D \cdot d(v_{\text{FLIP}}, v_{\text{OPT}})$ na $3 \cdot D \cdot d(v_{\text{FLIP}}, v'_{\text{OPT}})$. Z warunku trójkąta otrzymujemy:

$$\begin{aligned}\Delta\Phi &= 3 \cdot D[d(v'_{\text{FLIP}}, v'_{\text{OPT}}) - d(v'_{\text{FLIP}}, v_{\text{OPT}})] \\ &\leq 3 \cdot D[d(v_{\text{OPT}}, v'_{\text{OPT}})] \\ &\leq 3 \cdot \text{OPT} \ .\end{aligned}$$

Co kończy dowód (8) dla drugiego etapu. ■

Powiedzieć, że FLIP jest konkurencyjny również przeciwko adwersarzowi adaptującemu się.

Twierdzenie 10.2. *Załóżmy, że ALG jest R-konkurencyjny przeciwko adwersarzowi adaptującemu się. Załóżmy, ze istnieje algorytm ALG$'$, który jest R'-konkurencyjny przeciwko adwersarzowi nieświadomemu. Wtedy istnieje deterministyczny $(R \cdot R')$-konkurencyjny algorytm.*

10.2. Algorytm deterministyczny Move-To-Min

Rozważmy teraz następujący algorytm deterministyczny MOVE-TO-MIN (MTM). Algorytm ten dzieli całą sekwencję na fazy długości D. W każdej fazie algorytm przebywa w jednym wierzchołku, który oznaczamy P_{MTM} a pod koniec fazy przenosi się do tzw. *centrum grawitacji*, v^*. Wierzchołek v^* jest wierzchołkiem w którym byłoby najlepiej przebywać w danej fazie, tj. jest wirzchołkiem minimalizującym sumę $\sum_{i=1}^{D} d(v^*, \sigma_i)$.

Twierdzenie 10.3. *Algorytm MTM jest 7-konkurencyjny.*

Aby pokazać powyższe twierdzenie, zdefiniujemy funkcję potencjału równą $2 \cdot D \cdot d(P_{\text{MTM}}, P_{\text{OPT}})$ i pokażemy że zamortyzowany koszt w pojedynczej fazie f jest ograniczony.

Poniżej koncentrujemy się na pojedynczej fazie f. Numerujemy kroki w tej fazie od 1 do D. Wprowadzimy dodatkowe oznaczenie, które zilustrowane zostało na Rysunku 2. Mianowicie oznaczamy wierzchołek w którym OPT ma plik na początku kroku j przez a_{j-1}, a wierzchołek w którym OPT ma plik na końcu kroku j przez a_j. W szczególności a_0 i a_D są wierzchołkami, w których OPT ma plik odpowiednio na początku i końcu fazy.

Na początku pokażemy następujący pomocniczy lemat. Mówi on, że dolnym ograniczeniem na koszt algorytmu optymalnego jest koszt algorytmu, który całą fazę spędzi w dowolnym wierzchołku a_ℓ.

Lemat 10.4. *Dla dowolnej fazy f i dowolnego kroku $0 \leq \ell \leq D$, zachodzi* $\textsc{Opt}(f) \geq \sum_{i=1}^{D} d(a_\ell, \sigma_i)$.

Dowód. Zgodnie z oznaczeniami z rysunku koszt algorytmu optymalnego wynosi $\sum_{i=1}^{D}(d(a_{i-1}, \sigma_i)+D\cdot d(a_{i-1}, a_i))$. W powyższej sumie każda z odległości pomiędzy kolejnymi a_i jest liczona D razy. Z nierówności trójkąta otrzymujemy, że

$$\begin{aligned}\textsc{Opt}(f) &= \sum_{i=1}^{D} (d(a_{i-1}, \sigma_i) + D \cdot d(a_{i-1}, a_i)) \\ &\geq \sum_{i=1}^{D} (d(a_{i-1}, \sigma_i) + d(a_{i-1}, a_\ell)) \\ &\geq \sum_{i=1}^{D} d(a_\ell, \sigma_i) \ .\end{aligned}$$

■

Podzielmy koszt $\textsc{Mtm}(f)$ na dwie składowe: $\textsc{Mtm}^{\text{REQ}}(f)$ jest kosztem obsługi żądań w fazie f, a $\textsc{Mtm}^{\text{MOVE}}(f)$ jest kosztem przenosin pliku do wierzchołka v^*. Dodatkowo definiujemy $\Phi_{\text{B}}(f)$ i $\Phi_{\text{F}}(f)$ jako potencjał odpowiednio na początku i końcu fazy f. Należy zatem pokazać, że prawdziwe jest następujące stwierdzenie.

Lemat 10.5. *Dla dowolnej fazy f zachodzi*

$$\textit{Mtm}^{\text{REQ}}(f) + \textit{Mtm}^{\text{MOVE}}(f) + \Phi_{\text{F}}(f) \leq \Phi_{\text{B}}(f) + 7 \cdot \textit{Opt}(f).$$

Dowód. Z warunku trójkąta oraz Lematu 10.4 wynikają następujące nierówności:

$$\begin{aligned}\textsc{Mtm}^{\text{REQ}}(f) &= \sum_{i=1}^{D} d(P_{\text{MTM}}, \sigma_i) \leq \sum_{i=1}^{D} (d(P_{\text{MTM}}, a_0) + d(a_0, \sigma_i)) && (9)\\ &\leq D \cdot d(P_{\text{MTM}}, a_0) + \sum_{i=1}^{D} (a_0, \sigma_i) \leq \Phi_B/2 + \textsc{Opt}(f) \\ \textsc{Mtm}^{\text{MOVE}}(f) &= D \cdot d(P_{\text{MTM}}, v^*) \leq D \cdot d(P_{\text{MTM}}, a_0) + D \cdot d(a_0, v^*) && (10)\\ &\leq \Phi_B/2 + D \cdot d(a_0, v^*) \\ \Phi_F &= 2 \cdot D \cdot d(a_D, v^*) && (11)\end{aligned}$$

Wystarczy zatem ograniczyć $d(a_\ell, v^*)$ dla $0 \leq \ell \leq D$.

$$D \cdot d(a_\ell, v^*) \ = \sum_{i=1}^{D} d(a_\ell, v^*) \leq \sum_{i=1}^{D} d(a_\ell, \sigma_i) + \sum_{i=1}^{D} d(v^*, \sigma_i) \ .$$

Ponieważ v^* został wybrany jako wierzchołek który minimalizuje sumaryczną odległość od żądań w całej fazie, otrzymujemy:

$$D \cdot d(a_\ell, v^*) \ \leq 2 \cdot \sum_{i=1}^{D} d(a_\ell, \sigma_i) \leq 2 \cdot \textsc{Opt}(f)$$

Podstawiając to ograniczenie do nierówności (9), (10) i (11), otrzymujemy tezę lematu. ■

Dowód twierdzenia 10.3. Weźmy dowolną sekwencję σ i jej podział na fazy $\sigma = (f_1, f_2, \ldots, f_k, f_{k+1})$. Ostatnia faza ma być może mniej niż D kroków. Zauważmy, że zamortyzowany koszt w dowolnej z faz może zostać ograniczony przez funkcję zależną D i średnicy grafu G. Oznaczmy to ograniczenie przez β. Otrzymujemy wtedy

$$\begin{aligned}\textsc{Mtm}(\sigma) + \Delta\Phi(\sigma) &= \sum_{i=1}^{k} (\textsc{Mtm}(f_i) + \Delta\Phi(f_i)) + \textsc{Mtm}(f_{k+1}) + \Delta\Phi(f_{k+1}) \\ &\leq \sum_{i=1}^{k} (7 \cdot \textsc{Opt}(f_i)) + \beta \\ &\leq 7 \cdot \textsc{Opt}(\sigma) + \beta\end{aligned}$$

Ponieważ $\Delta\Phi(\sigma)$ jest nieujemne otrzymujemy $\textsc{Mtm}(\sigma) \leq 7 \cdot \textsc{Opt}(\sigma) + \beta$, co kończy dowód konkurencyjności algorytmu $\textsc{Mtm}$. ■

10.2.1. Algorytm Move-To-Local-Min

Na to nie starczy czasu, ale zostawiam w notatkach.

Zauważmy, że MTM kiepsko radzi sobie z sytuacją, w której jest wiele minimów będących kandydatami na v^*. Wtedy MTM wybiera jedno z nich, podczas gdy mógłby wybierać najbliższe. Warto również wybrać wierzchołek prawie minimalny jeśli jest on znacznie bliżej aktualnej pozycji pliku niż globalne minimum. Prowadzi nas to do algorytmu MOVE-TO-LOCAL-MIN (MTLM), który różni się od MTM tylko wyborem v^*. MTLM wybiera na v^* ten wierzchołek v, który minimalizuje sumę $2 \cdot \sum_{i=1}^{D} d(v, \sigma_i) + D \cdot d(P_{\text{MTLM}}, v)$.

Twierdzenie 10.6. *MTLM jest 5-konkurencyjny.*

Dowód. Wystarczy pokazać odpowiednik Lematu 10.5, reszta dowodu jest identyczna jak w przypadku algorytmu MTM. Ograniczenie na $\text{MTLM}^{\text{REQ}}(f)$ jest identyczne, tj.

$$\text{MTLM}^{\text{REQ}}(f) \leq \Phi_{\text{B}}/2 + \text{OPT}(f) \ .$$

Z kolei

$$\begin{aligned}
\text{MTLM}^{\text{MOVE}}(f) + \Phi_{\text{F}} &\leq D \cdot d(P_{\text{MTLM}}, v^*) + 2 \cdot D \cdot d(a_D, v^*) \\
&\leq D \cdot d(P_{\text{MTLM}}, v^*) + 2 \cdot \sum_{i=1}^{D} d(v^*, \sigma_i) + 2 \cdot \sum_{i=1}^{D} d(a_D, \sigma_i) \\
&\leq D \cdot d(P_{\text{MTLM}}, v^*) + 2 \cdot \sum_{i=1}^{D} d(v^*, \sigma_i) + 2 \cdot \text{OPT}(f)
\end{aligned}$$

Z minimalności v^* w pierwszych dwóch składnikach powyżej możemy zamienić v^* na a_0 otrzymując

$$\begin{aligned}
\text{MTLM}^{\text{MOVE}}(f) + \Phi_{\text{F}} &\leq D \cdot d(P_{\text{MTLM}}, a_0) + 2 \cdot \sum_{i=1}^{D} d(a_0, \sigma_i) + 2 \cdot \text{OPT}(f) \\
&\leq \Phi_{\text{B}}/2 + 4 \cdot \text{OPT}(f) \ .
\end{aligned}$$

■

11. Funkcje pracy: metryczne systemy zadań

W.9A

METRYCZNE SYSTEMY ZADAŃ. Dany jest skończony zbiór stanów S i metryka d na tym zbiorze. Dany jest również stan początkowy $s_0 \in S$. W każdym kroku t algorytm otrzymuje wektor kar r_t, określający ile trzeba zapłacić za bycie w danym stanie. Algorytm zmienia swój stan z s_{t-1} do s_t płacąc $d(s_{t-1}, s_t)$ (możliwe, że $s_t = s_{t-1}$) a następnie płaci $r_t(s_t)$. Należy zminimalizować sumaryczny koszt.

Pokażemy teraz $(2n-1)$-konkurencyjny algorytm dla metrycznych systemów zadań (dla dowolnej metryki).

- Bez straty ogólności możemy pozwolić, żeby OPT zmieniał stan również na końcu kroku (tj. po zapłaceniu za żądanie).
- Bez straty ogólności, OPT może zmieniać stan na końcu kroku 0.
- Bez straty ogólności OPT zmienia stan tylko na końcu kroku.

Dla dowolnego kroku t i stanu x definiujemy funkcję pracy $w_t(x)$ jako minimalny koszt obsłużenia sekwencji wejściowej do kroku t włącznie i zakończenia w stanie x. Wtedy $w_0(s) = d(s_0, s)$ oraz

$$w_{t+1}(s) = \min_{x \in S}\{w_t(x) + r_{t+1}(x) + d(x, s)\} \ . \tag{12}$$

Zauważmy, że dla sekwencji σ o długości m zachodzi $\text{OPT}(\sigma) = \min_{s \in S} w_m(s)$ (w szczególności OPT nie przesuwa się po ostatnim żądaniu).

11.1. Intuicje

Ewolucja funkcji pracy.

Obserwacja 11.1. *Dla każdego czasu t i pary stanów x i y zachodzi $|w_t(x) - w_t(y)| \le d(x, y)$*

Obserwacja 11.2. *Dla każdego czasu t i stanu x zachodzi $w_{t+1}(x) \le w_t(x) + r_{t+1}(x)$.*

O ewolucji funkcji w w czasie wygodnie myśleć w ten sposób, że w chwili $t+1$ obliczamy $w_{t+1}(\cdot)$ w następujący sposób. Najpierw obliczamy $\widetilde{w}_{t+1}(x) = w_t(x) + r_{t+1}(x)$. Następnie jeśli jakieś wartości $\widetilde{w}_{t+1}$ są za duże, to je przycinamy: jeśli istnieje para y i x taka, że $\widetilde{w}_{t+1}(y) > \widetilde{w}_{t+1}(x) + d(x, y)$ to przypisujemy $\widetilde{w}_{t+1}(y) \leftarrow \widetilde{w}_{t+1}(x) + d(x, y)$. Następnie wartości $\widetilde{w}_{t+1}(\cdot)$ stają się wartościami $w_{t+1}(\cdot)$

Kombinacja liniowa funkcji pracy. Na końcu kroku t wartość optymalnego rozwiązania to $\min_{s \in S} w_t(s)$. Ale biorąc pod uwagę Obserwację 11.1 zamiast minimum można rozważać dowolną kombinację liniową $\sum_{i=1}^{n} p_i \cdot w_t(v_i)$ gdzie $\sum_{i=1}^{n} p_i = 1$. Kombinacja taka jest odległa od OPT o co najwyżej addytywny czynnik zależny od średnicy grafu.

Wynika z tego, że niebezpieczne dla algorytmu są takie sytuacje, że ponosi on koszt, choć cała funkcja pracy pozostaje bez zmian. Albo gdy nie potrafimy powiązać jego kosztu ze zmianami funkcji pracy.

Nazwijmy stan x *górnym* w chwili t jeśli $w_t(x) = w_t(y) + d(x, y)$ dla pewnego $y \neq x$. Problem pojawia się, jeśli algorytm jest i pozostaje w takim górnym stanie x i w kroku $t+1$ zachodzi $r_{t+1}(x) > 0$ i $r_{t+1}(y) = 0$, bo wtedy algorytm ponosi koszt $r_{t+1}(x)$ natomiast funkcja pracy może się nie zmienić. Nasz algorytm będzie zatem pilnować, żeby kroku t nigdy nie kończyć w górnym stanie.

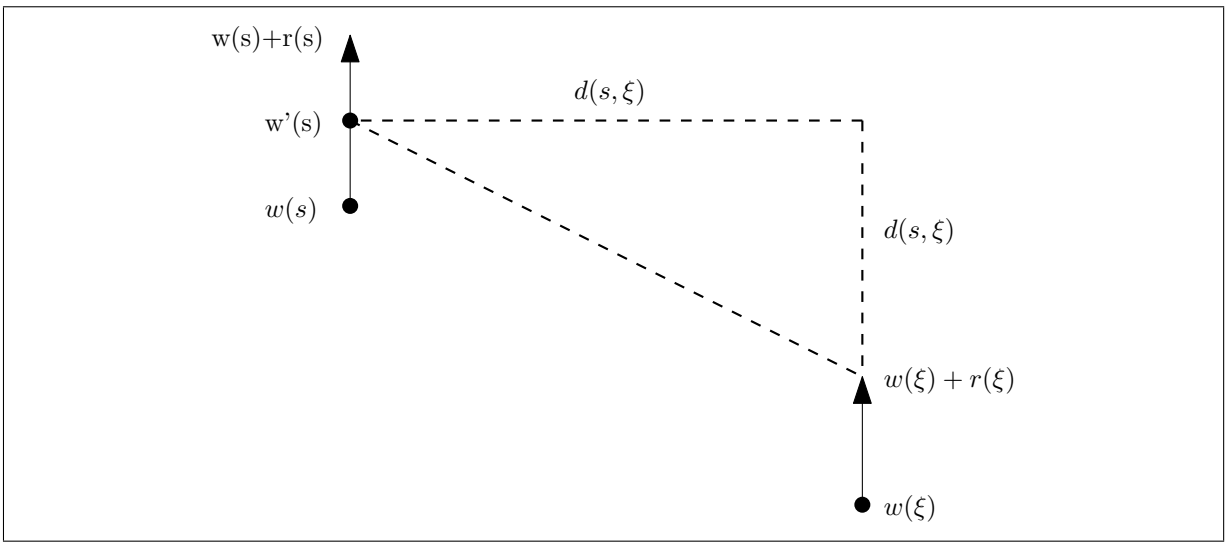
11.2. Algorytm WFA

Zdefiniujemy algorytm WFA (*work function algorithm*) dla kroku $t+1$. Dla uproszczenia notacji w tej podsekcji będziemy zakładać, że $w = w_t$, $w' = w_{t+1}$, $s = s_t$, $s' = s_{t+1}$ oraz $r = r_{t+1}$.

Algorytm jest w stanie s na początku kroku. Następnie po obliczeniu $w'(s)$ sprawdza jakie jest ξ, które „wyznaczyło" wartość $w'(s)$, czyli ξ spełniające

$$w'(s) = w(\xi) + r(\xi) + d(\xi, s)$$

Może się wydarzyć, ze $\xi = s$. Jeśli jest wiele takich ξ wybieramy jedno z nich. Następnie (przed obsłużeniem żądania) algorytm zmienia stan na ξ, tj. $s' = \xi$.



11.3. Konkurencyjność WFA

W.9B

Sytuację dla $\xi \neq s$ ilustruje poniższy obrazek.

Jaka jest wartość $w'(\xi)$? Z definicji funkcji pracy $w'(\xi) \leq w(\xi) + r(\xi)$. Natomiast z Obserwacji 11.1 wynika, że $w'(\xi) \geq w'(s) - d(\xi, s) = w(\xi) + r(\xi)$. Stąd

$$w'(\xi) = w(\xi) + r(\xi) \ .$$

Czyli po przenosinach do wierzchołka ξ płacimy tam tyle samo o ile wzrasta tam funkcja pracy ($r(\xi)$). Taka sama własność (bezpośrednio) zachodzi jeśli $\xi = s$.

Lemat 11.3. $W_{FA}(t+1) = w_{t+1}(s_t) - w_t(s_{t+1})$

Dowód. Dowód obrazkowy: jeśli algorytm nie zmienia stanu ($s_{t+1} = s_t$), to płaci $r(s_t) = w_{t+1}(s_t) - w_t(s_t)$. W przeciwnym przypadku płaci $d(s_t, s_{t+1}) + r(s_{t+1}) = w_{t+1}(s_t) - w_t(s_{t+1})$ (patrz rysunek).

Dowód syntaktyczny: $\text{WFA}(t+1) = d(s_t, s_{t+1}) + r(s_{t+1}) = w_{t+1}(s_t) - w_t(s_{t+1})$ (gdzie ostatnia równość wynika z definicji algorytmu). ■

Twierdzenie 11.4. *Algorytm WFA jest $2n$-konkurencyjny.*

Dowód. Posumujmy koszt algorytmu po m krokach

$$\begin{aligned}
\text{WFA}(\sigma) = & \ w_1(s_0) - w_0(s_1) \\
& + w_2(s_1) - w_1(s_2) \\
& + w_3(s_2) - w_2(s_3) \\
& + \ldots \\
& + w_{m-2}(s_{m-3}) - w_{m-3}(s_{m-2}) \\
& + w_{m-1}(s_{m-2}) - w_{m-2}(s_{m-1}) \\
& + w_m(s_{m-1}) - w_{m-1}(s_m)
\end{aligned}$$

Traktując pierwszy wyraz osobno, pomijając (ujemny) ostatni wyraz, oraz grupując dostajemy

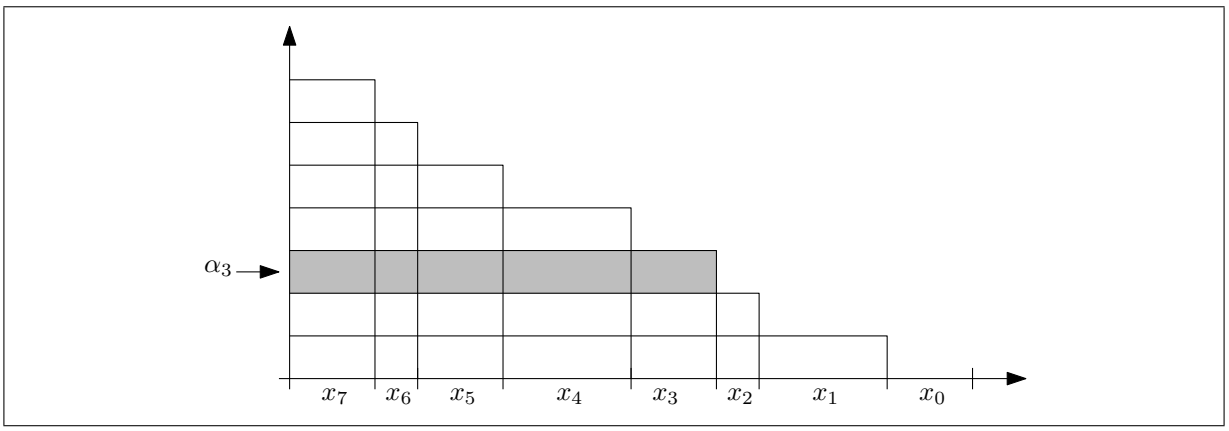
$$\begin{aligned}
\text{WFA}(\sigma) &\leq w_1(s_0) + \sum_{t=0}^{m-2} \left(w_{t+2}(s_{t+1}) - w_t(s_{t+1}) \right) \\
&= \Delta w_1(s_0) + \sum_{t=0}^{m-2} \left(\Delta w_{t+2}(s_{t+1}) + \Delta w_{t+1}(s_{t+1}) \right) \ ,
\end{aligned}$$

gdzie $\Delta w_t(s) = w_t(s) - w_{t-1}(s)$. Zauważmy, że w powyższym wyrażeniu $w_t(\cdot)$ pojawia się co najwyżej 2 razy

dla każdego v_i a stąd

$$\begin{aligned}
\textsc{Wfa}(\sigma) &\leq \sum_{i=1}^{n} 2 \sum_{t=1}^{T} \Delta w_t(s_i) \\
&= \sum_{i=1}^{n} 2 w_T(s_i) \\
&\leq \sum_{i=1}^{n} 2(\textsc{Opt}(\sigma) + O(\textsc{diam}(G)) \\
&= 2n \cdot \textsc{Opt}(\sigma) + O(n \cdot \textsc{diam}(G)) \,.
\end{aligned}$$

Licząc dokładniej można pokazać, że Wfa jest $(2n-1)$-konkurencyjny. ■



Rysunek 3: Zysk algorytmu BAL: ilustracja dla $b = 7$. Niech x_i = liczba graczy, którzy wydali i dolarów. Niech α_j = ile odsłon zostało kupione j-tym dolarem.

12. Adwords

PROBLEM ADWORDS. Dany jest zbiór słów kluczowych Q i zbiór n graczy (reklamodawców). Gracz i ma budżet b_i i definiuje ceny $c_{i,q}$ = ile chce zapłacić, za to że przy próbie wyszukania słowa kluczowego $q \in Q$ wyświetli się jego reklama.

Input: ciąg $q_1, q_2, \ldots \in Q$
Algorytm dla q_j wybiera gracza i i zyskuje $c_{i,q}$. Nie można przekraczać budżetu graczy, chcemy zmaksymalizować swój zysk.

Będziemy rozpatrywać bardzo uproszczony przypadek:

- $b_1 = b_2 = \ldots = b_n = b$.
- $c_{i,q} \in \{0, 1\}$

Algorytm BAL: Dla słowa kluczowego q_j, spośród zainteresowanych graczy (takich z $c_{i,q_j} = 1$ wybierz zainteresowanego gracza, który dotąd wydał najmniej.
Dodatkowo będziemy zakładać, że:

- b jest duże (istotne: konkurencyjność będzie zbiegać do dobrego współczynnika dla $b \to \infty$)
- OPT $= n \cdot b$ (mało istotne, upraszcza analizę)

12.1. Notacje i obserwacje

Zakładamy, że gracz ma ponumerowane dolary (od 1 do b) i wydaje je po kolei. Przyjmijmy notację z Rysunku 3. Wtedy

$$\text{BAL} = \sum_{j=1}^{b} \alpha_j \quad \text{oraz} \quad \text{OPT} = n \cdot b$$

Lemat 12.1. *Dla dowolnego $i \in \{0, \ldots, b-1\}$ zachodzi $b \cdot \sum_{j=0}^{i} x_j \leq \sum_{j=1}^{i+1} \alpha_j$*

Dowód. Warto zaznaczyć to jako obszary na rysunku. Niech X_j będzie zbiorem graczy, którzy wydają j dolarów ($|X_j| = x_j$). Popatrzmy na te słowa q które OPT przypisał graczom z $\bigcup_{j=1}^{i} X_j$. Takich słów jest oczywiście $\sum_{j=1}^{i} b \cdot x_j$.
Każde z tych słów jest obsłużone przez BAL i co więcej obsłużone dolarem o indeksie $\leq i + 1$. Dlaczego? Załóżmy że q jest w OPT obsłużone przez gracza $g \in X_k$ ($k \leq i$). Wtedy w momencie gdy q pojawia się, w rozwiązaniu BAL, gracz g wydał co najwyżej k dolarów (bo tyle wydał w ogóle): w rozwiązaniu BAL ma kto kupić q i ci istnieją gracze, którzy są q zainteresowani i wydali do tej pory $\leq k$ dolarów. Więc BAL płaci za q dolarem o indeksie co najwyżej $k + 1$. ■

Wniosek 12.2. *Bal jest 2-konkurencyjny.*

Dowód. Z Lematu 12.1 dla $i = b - 1$ mamy

$$\textsc{Bal} = \sum_{j=1}^{b} \alpha_j \geq b \cdot \sum_{i=1}^{b-1} x_j = b \cdot (n - x_b) = b \cdot n - b \cdot x_b$$

Stąd

$$2 \cdot \textsc{Bal} \geq \textsc{Bal} + b \cdot x_b \geq b \cdot n = \textsc{Opt}$$

■

12.2. Lepsza analiza

Przekształćmy Lemat 12.1 wykorzystując definicję α_j. Otrzymamy, że dla dowolnego $i \in \{0, \ldots, b-1\}$ zachodzi

$$b \cdot \sum_{j=0}^{i} x_j \leq \sum_{j=1}^{i+1} \alpha_j = n \cdot (i+1) - \sum_{j=0}^{i} x_j \cdot (i+1-j)$$

a stąd

$$\sum_{j=0}^{i} (b+i+1-j) \cdot x_j \leq n \cdot (i+1) \tag{13}$$

Z drugiej strony zysk Bal to $\textsc{Bal} = n \cdot b - P(x_0, \ldots, x_{b-1})$, gdzie

$$P(x_0, \ldots, x_{b-1}) = \sum_{j=0}^{b-1} (b-j) \cdot x_j$$

Chcemy teraz oszacować $P(x_0, \ldots, x_{b-1})$ od góry. Rozważmy to jako problem optymalizacyjny: zmaksymalizować $P(x_0, \ldots, x_{b-1})$ przy założeniu, że nierówności (13) zachodzą. Po rozpisaniu te nierówności wyglądają tak:

$$\begin{array}{rrrrl}
(b+1) \cdot x_0 & & & & \leq n \\
(b+2) \cdot x_0 + (b+1) \cdot x_1 & & & & \leq 2 \cdot n \\
(b+3) \cdot x_0 + (b+2) \cdot x_1 & +(b+1) \cdot x_2 & & & \leq 3 \cdot n \\
\ldots & \ldots & \ldots & \ldots & \\
(2b-1) \cdot x_0 + (2b-2) \cdot x_1 + (2b-3) \cdot x_2 + \ldots + (b+1) \cdot x_{b-2} & & & & \leq (b-1) \cdot n \\
2b \cdot x_0 + (2b-1) \cdot x_1 + (2b-2) \cdot x_2 + \ldots + (b+2) \cdot x_{b-2} + (b+1) \cdot x_{b-1} & & & & \leq b \cdot n
\end{array}$$

Można te nierówności uruchomić w dowolnym LP solverze i wywnioskować, że $P^* = \max_{x_i} P(x_0, \ldots, x_{b-1})$ wynosi około $n \cdot b/\mathrm{e}$. Udowodnienie tego formalnie jest trudniejsze, bo trzeba pokazać, że taka relacja zachodzi dla dowolnego wyboru $x_0, \ldots, x_{b-1}$ przez adwersarza.

Napiszmy program dualny. Polega on na minimalizacji wyrażenia $D(y_0, \ldots, y_{b-1}) = \sum_{j=0}^{b-1} n \cdot (j+1) \cdot y_j$ przy warunkach:

$$\begin{array}{rrrrl}
(b+1) \cdot y_{b-1} & & & & \geq 1 \\
(b+2) \cdot y_{b-1} + (b+1) \cdot y_{b-2} & & & & \geq 2 \\
(b+3) \cdot y_{b-1} + (b+2) \cdot y_{b-2} & + (b+1) \cdot y_{b-3} & & & \geq 3 \\
\ldots & \ldots & & & \ldots \\
2b \cdot y_{b-1} + (2b-1) \cdot y_{b-2} + (2b-2) \cdot y_{b-3} + \ldots + & (b+1) \cdot y_0 & & & \geq b
\end{array}$$

Niech $D^* = \min_{y_i} D(y_0, \ldots, y_{b-1})$. Ze słabego twierdzenia o dualności dostajemy, że dla dowolnych x_i i y_i zachodzi:

$$P(x_0, \ldots, x_{b-1}) \leq P^* \leq D^* \leq D(y_0, \ldots, y_{b-1})$$

Czyli jeśli ustalimy dowolne dopuszczalne rozwiązanie $\{y_i\}_{i=0}^{b-1}$, to $D(y_0,\ldots,y_{b-1})$ będzie górnym ograniczeniem na wartość $P(x_0,\ldots,x_{b-1})$ dla *każdego wyboru* $\{x_i\}_{i=0}^{b-1}$!

Jak znaleźć odpowiednie y_i? Można spróbować zamienić wszystkie nierówności w programie dualnym na równości i rozwiązać. Otrzymujemy wtedy

$$y_i = \frac{1}{b+1}\cdot\left(\frac{b}{b+1}\right)^{b-i-1}.$$

Łatwo zweryfikować że są one dopuszczalnym rozwiązaniem. Wtedy oznaczając $q = b/(b+1)$ mamy

$$\begin{aligned} D(y_0,\ldots,y_{b-1}) &= \sum_{j=0}^{b-1} n\cdot(j+1)\cdot y_j = \frac{n}{b+1}\cdot\sum_{j=0}^{b-1}(j+1)\cdot q^{b-(j+1)} \\ &= \frac{n}{b+1}\cdot\sum_{j=0}^{b-1}(b-j)\cdot q^j = \frac{n}{b+1}\cdot\left(b\cdot\sum_{j=0}^{b-1}q^j - \sum_{j=0}^{b-1} j\cdot q^j\right) \\ &= \frac{n}{b+1}\cdot\left(b\cdot\frac{1-q^b}{1-q} - \frac{q\cdot\big((b-1)\cdot q^b - b\cdot q^{b-1}+1\big)}{(1-q)^2}\right) \end{aligned}$$

Zauważamy teraz, że $q/(1-q) = b$ a następnie $(b+1)\cdot(1-q) = 1$ otrzymując

$$\begin{aligned} D(y_0,\ldots,y_{b-1}) &= n\cdot\frac{b}{b+1}\cdot\left(\frac{1-q^b}{1-q} - \frac{(b-1)\cdot q^b - b\cdot q^{b-1}+1}{1-q}\right) \\ &= n\cdot b\cdot\big(1-q^b-(b-1)\cdot q^b + b\cdot q^{b-1}-1\big) \\ &= n\cdot b\cdot\big(b\cdot q^{b-1}\cdot(1-q)\big) = n\cdot b\cdot q^b \\ &= n\cdot b\cdot\left(\frac{b}{b+1}\right)^b \end{aligned}$$

A zatem zysk algorytmu to co najmniej $\textsc{Bal} = n\cdot b - n\cdot b\cdot(b/(b+1))^b$. Zauważmy, że dla dużych b, $D(y_0,\ldots,y_{b-1})$ to wyrażenie zbiega do $n\cdot b\cdot(1-1/\mathrm{e}) = n\cdot b\cdot((\mathrm{e}-1)/\mathrm{e})$, czyli współczynnik konkurencyjności Bal wynosi asymptotycznie $\mathrm{e}/(\mathrm{e}-1)\approx 1.582$.

13. Przybliżanie grafów za pomocą drzew

Prezentacja w powerpoincie

A. Metoda obserwacji rozkładu prawdopodobieństwa

Choć adwersarz nie zna bitów losowych algorytmu, może obliczyć jego rozkład prawdopodobieństwa.

Twierdzenie A.1. *Konkurencyjność każdego zrandomizowanego algorytmu dla problemu pamięci podręcznej wynosi co najmniej H_k.*

Ustalmy dowolny algorytm zrandomizowany ALG. Pokażemy, jak adwersarz może wygenerować sekwencję σ o dowolnie dużym koszcie dla algorytmu optymalnego, taką że $\mathbf{E}[\text{ALG}(\sigma)]/\text{OPT}(\sigma)$ będzie dowolnie bliskie H_k. W dowodzie będziemy wykorzystywać tylko strony o numerach $1, 2, 3, \ldots, k, k+1$. Konstrukcja σ składa się z prefiksu $1, 2, 3, \ldots, k, k+1$ po którym następuje ciąg faz. Dla każdej fazy f pokażemy, że $\mathbf{E}[\text{ALG}(f)] \geq H_k$ oraz $\text{OPT}(f) = 1$. Faza f składać się będzie z k podfaz $f_1, f_2, \ldots, f_k$.

Idea. Na początku podfazy f_j j elementów będziemy nazywać pomalowanymi. W podfazie f_j pytamy o te elementy wiele razy, zmuszając ALG do utrzymania ich w cache. Następnie pytamy o niepomalowany element x (taki, do którego odwołanie spowoduje u ALG jak największy koszt) i malujemy x.

Element pomalowany na początku podfazy f_1 to ten do którego odwołanie było na końcu poprzedniej fazy.

Przykład. Weźmy $k = 3$. Prefiks σ to `A B C D`. Pierwsza faza może wyglądać następująco

```
| {D}* B | {B,D}* A | {A,B,D}* C |
```

Zauważmy, że jeśli ALG w podfazie f_j ma wszystkie strony pomalowane, to istnieje taka niepomalowana strona, której nie ma z prawdopodobieństwem $1/(k+1-j)$. Zatem taki algorytm w fazie zapłaci $\sum_{j=1}^{k} 1/(k+1-j) = H_k$.

Górne ograniczenie na OPT. Ustalmy fazę f. Zauważmy, że faza poza ostatnim żądaniem (do q_f) składa się z k różnych stron. Dodatkowo q_f jest różny od ostatniego żądania w poprzedniej fazie. Zatem OPT przy obsłudze ostatniego żądania w poprzedniej fazie może wyrzucić q_f z cache, obsłużyć całą fazę f poza ostatnim żądniem bezstratnie i zapłacić 1 za q_f.

Dolne ograniczenie na ALG. Wystarczy pokazać, że w podfazie f_j możemy wymusić koszt $1/(k+1-j)$.

Dla dowolnej strony s niech p_s będzie prawdopodobieństwem, że ALG nie ma tej strony w pamięci podręcznej. Oczywiście $\sum_s p_s = 1$. Niech M będzie zbiorem stron pomalowanych na początku podfazy f_j; $|M| = j$. Rozważamy dwa przypadki

1. $\sum_{s \in M} p_s = 0$. Oznacza to, że algorytm ma wszystkie pomalowane strony w swojej pamięci podręcznej (dowolny algorytm zaznaczający miałby taką własność). Podfaza f_j to po prostu odwołanie do strony $a = \arg\max_s \{p_s\}$. Wtedy $\mathbf{E}[\text{ALG}] = p_a \geq 1/(k+1-|M|) = 1/(k+1-j)$.

2. $\sum_{s \in M} p_s > 0$. Niech $\varepsilon = \max_{m \in M} p_m$. Adwersarz wykonuje następującą pętlę: dopóki $\sum_{s \in M} p_s \geq \varepsilon$ i $\mathbf{E}[\text{ALG}(f_j)] < 1/(k+1-j)$, pytamy o najbardziej „prawdopodobną" pomalowaną stronę. Po wyjściu z pętli adwersarz pyta o $a = \arg\max_s \{p_s\}$.

   Po pierwsze zauważmy, że pętla zawsze się kończy: w każdym kroku istnieje strona $b \in M$, dla której $p_b \geq \sum_{s \in M} p_s/|M| > \varepsilon/|M|$, więc koszt ALG w końcu osiąga zadaną wartość.

   - Jeśli pętla kończy się, gdyż koszt $\mathbf{E}[\text{ALG}] \geq 1/(k+1-i)$, to ograniczenie zachodzi trywialnie.
   - Jeśli pętla kończy się, gdyż $\sum_{s \in M} p_s < \varepsilon$, to wtedy patrzymy na oczekiwany koszt ALG związany z odwołaniem do strony m i strony a:

$$p_m + p_a \geq \varepsilon + \frac{1-\varepsilon}{k+1-|M|} \geq \frac{1}{k+1-|M|} = \frac{1}{k+1-j} \ .$$

B. Zarządzanie plikami

W tej części rozważymy rozszerzenie problemu przenoszenia pliku. Kiedy pozbędziemy się założenia, że może być tylko jedna kopia pliku w sieci i rozróżnimy pomiędzy odwołaniami odczytu z pliku i zapisu do niego, otrzymamy następujący problem.

ZARZĄDZANIE PLIKIEM. Dany jest ciąg żądań $\text{REQ}(\sigma_t)$, gdzie $\text{REQ} \in \{\text{READ}, \text{WRITE}\}$ a σ_t jest jakimś wierzchołkiem. Niech Q oznacza zbiór wierzchołków w których algorytm ma kopię pliku. Przy żądaniu odczytu z pliku algorytm płaci za odległość między σ_t a najbliższym wierzchołkiem ze zbioru Q. Przy żądaniu zapisu algorytm musi uaktualnić wszystkie kopie, płaci zatem za koszt minimalnego drzewa Steinera opartego na wierzchołkach $Q \cup \{\sigma_t\}$. Po obsłudze żądania algorytm może skopiować plik z dowolnego wierzchołka u do dowolnego innego u', płacąc $D \cdot d(u, u')$. Można też za darmo usunąć kopię z dowolnego wierzchołka.

Zauważmy, że jeśli sekwencja składa się z samych żądań WRITE, to utrzymywanie więcej niż jednej kopii pliku nie ma sensu i cały problem staje się problemem przenoszenia pliku.

Z drugiej strony podproblem, w którym występują tylko żądania READ nazywamy problemem kopiowania pliku (usuwanie kopii pliku nie ma sensu). W tym podproblemie ma sens mówienie tylko o ścisłej konkurencyjności, bo koszt kopiowania pliku do wszystkich wierzchołków jest funkcją rozmiaru grafu i jest niezależny od wejścia.

Poniżej przedstawiamy algorytm COUNT dla problemu zarządzania plikiem w klikach (pełnych grafach z odległościami między każdą parą wierzchołków równą 1). Każdy wierzchołek v ma licznik c_v, początkowo ustawiony na 0. Każdy wierzchołek v wykonuje następujący program $\text{COUNT}(v)$:

while $c_v < D$ **:**
if (READ(v) i $v \notin Q$) or (WRITE(v) a „sieć oczekuje")
then c_v**++**
endfi
endwhile
Skopiuj plik do v

while $c_v > 0$ **:**
if WRITE(w) i $v \neq w$
then c_v**−**
endif
end-while
Jeśli masz ostatnią kopię pliku w sieci, przejdź w stan oczekiwania.
Skasuj plik z v.
Przejdź do punktu 1.

Twierdzenie B.1. *Algorytm* COUNT *jest* 3*-konkurencyjny dla problemu zarządzania plikiem*

Dowód. Po pierwsze zaobserwujmy, że tylko jeden procesor może być w stanie oczekiwania. Co więcej pozostałe wierzchołki wykonują wtedy punkt 1 (mówimy wtedy, że cała *sieć jest w stanie oczekiwania*).

Nazwijmy wykonanie kroków od 1 do 6 w wierzchołku v fazą wierzchołka v. Oczywiście fazy różnych wierzchołków niekoniecznie pokrywają się w czasie.

Niech Q będzie zbiorem wszystkich wierzchołków, które mają plik. W definicji problemu koszty zostały zdefiniowane globalnie. Dla celów analizy przypiszemy te koszty poszczególnym procesorom.

(i) Procesor, który zgłasza żądanie odczytu, płaci 1.

(ii) Jeśli sieć jest w stanie oczekiwania i procesor v zgłasza żądanie zapisu, v płaci 1.

(iii) Jeśli sieć nie jest w stanie oczekiwania i procesor v zgłasza żądanie zapisu, płacą wszystkie wierzchołki ze zbioru $Q \setminus \{v\}$.

(iv) Jeśli plik jest kopiowany, D płaci wierzchołek, który otrzymuje plik.

Weźmy dowolny wierzchołek v. Przy powyższej definicji opłat, każdej zapłacie o 1 odpowiada zwiększenie lub zmniejszenie licznika c_v o 1. Zatem v płaci dokładnie $3 \cdot D$ w pojedynczej fazie. Wyjątkiem jest pierwsza faza, w której wierzchołki zaczynające z plikiem płacą tylko D.

Załóżmy, że w rozwiązaniu optymalnym, OPT płaci za usuwanie pliku a nie za jego kopiowanie. Przez takie założenie koszt pojedynczego wierzchołka może zostać zmodyfikowany co najwyżej o D, zatem koszt całego rozwiązania optymalnego zostanie zaburzony o co najwyżej $n \cdot D$. Stała ta nie ma znaczenia przy odpowiednio długiej sekwencji wejściowej.

Teraz obliczmy jaki jest koszt algorytmu optymalnego związany z pojedynczym wierzchołkiem v w krokach, które stanowią pojedynczą fazę wierzchołka v. Jeśli OPT usuwa plik z v w danej fazie, to wtedy jego koszt to co najmniej D. W przeciwnym przypadku OPT nie usuwa pliku z v i mamy dwa podprzypadki. Jeśli ma plik w v przez całą fazę, to wtedy płaci 1 za każde z D żądań w kroku 3. Jeśli nie ma pliku w v przez całą fazę, to wtedy płaci 1 za każde z D żądań w kroku 1. W każdym z tych przypadków koszt przypisany wierzchołkowi v w rozwiązaniu optymalnym wynosi co najmniej D. ■

C. K-Server: Algorytm Balance

W tej sekcji pokażemy, że algorytm BALANCE jest k-konkurencyjny dla dowolnych $k+1$-punktowych przestrzeni metrycznych. Definiujemy $D(v_i)$ jako długość drogi, którą przebył serwer obecnie znajdujący się w wierzchołku v_i (jest tylko jeden taki serwer). Najpierw pokażemy następujący lemat.

Lemat C.1. *Dla dowolnych v_i i v_j, w których są jakieś serwery zachodzi*

$$|D(v_i) - D(v_j)| \leq d(v_i, v_j)$$

Dowód. Udowodnimy ten lemat przez indukcję względem liczby kroków. Na początku $D(\cdot) \equiv 0$ i nierówność jest trywialnie spełniona. Załóżmy, że nierówność jest spełniona do początku pewnego kroku i pokażemy, że zachodzi też na jego końcu. Bez straty ogólności możemy założyć, że na początku kroku serwera nie ma w punkcie v_0. Jeśli żądanie przychodzi z wierzchołka innego niż v_0, BAL nie przesuwa żadnego ze swoich serwerów; możemy zatem założyć, że żądanie jest w wierzchołku v_0. Załóżmy, że BAL przesuwa do v_0 serwer, który był do tej pory w v_ℓ.

Dla porządku będziemy oznaczać wartości D na końcu kroku przez D'. Zauważmy, że dla $i \neq 0, \ell$ wartość $D'(v_i)$ jest taka sama jak $D(v_i)$ oraz $D'(v_0) = D(v_\ell) + d(v_0, v_\ell)$.[5] Zatem wystarczy pokazać, że na końcu kroku $|D'(v_0) - D'(v_j)| \leq d(v_0, v_j)$ zachodzi dla dowolnego $j \neq 0, \ell$.

Z założenia indukcyjnego oraz nierówności trójkąta otrzymujemy

$$D'(v_j) - D'(v_0) \;=\; D(v_j) - D(v_\ell) - d(v_0, v_\ell) \;\leq\; d(v_j, v_\ell) - d(v_0, v_\ell) \;\leq\; d(v_j, v_0) \;. \tag{14}$$

Z drugiej strony

$$D'(v_0) - D'(v_j) \;=\; D(v_\ell) + d(v_0, v_\ell) - D(v_j) \;.$$

Ponieważ v_ℓ został wybrany przez algorytm BAL do przesunięcia, musiała zachodzić nierówność $D(v_\ell) + d(v_\ell, v_0) \leq D(v_j) + d(v_j, v_0)$. Zatem

$$D'(v_0) - D'(v_j) \;\leq\; D(v_j) + d(v_0, v_j) - D(v_j) = d(v_0, v_j) \;. \tag{15}$$

Z nierówności 14 i 15 wynika teza lematu. ∎

Zauważmy, że BAL zawsze pokrywa swoimi serwerami k wierzchołków. Niepokryty wierzchołek nazywamy *dziurą*. Wierzchołek, w którym BAL ma obecnie dziurę będziemy oznaczać przez CURR a wierzchołek, w którym BAL miał poprzednio dziurę nazywamy PREV. Oznacza to, że ostatnim przesunięciem jakie BAL wykonał były przenosiny serwera z wierzchołka CURR do wierzchołka PREV. Przyglądając się bliżej nierówności 14 można zauważyć następującą rzecz.

Obserwacja C.2. *Dla dowolnego v_j, w którym jest serwer (tj. różnym od CURR) zachodzi*

$$D(v_j) - D(\textsc{prev}) \leq d(v_j, \textsc{curr}) - d(\textsc{curr}, \textsc{prev}) \;.$$

Dowód. Jeśli $v_j = \text{PREV}$, to powyższa nierówność jest spełniona trywialnie. W przeciwnym przypadku obserwacja wynika z nierówności 14 po podstawieniu PREV pod v_ℓ i CURR pod v_0. ∎

Twierdzenie C.3. *Algorytm BAL jest k-konkurencyjny dla dowolnych $k+1$-punktowych przestrzeni metrycznych.*

Dowód. Zauważmy, że istnieje algorytm optymalny, który jest leniwy, zatem OPT ma też jedną dziurę. Niech $S = \sum_{v_i \neq \text{CURR}} D(v_i)$. Zdefiniujmy następującą funkcję potencjału

$$\Phi = \begin{cases} k \cdot (D(\text{PREV}) - d(\text{CURR}, \text{PREV})) - S & \text{jeśli dziura OPT jest w CURR} \;, \\ k \cdot D(v) - S & \text{jeśli dziura OPT jest w } v \neq \text{CURR} \;. \end{cases}$$

Zauważmy, że funkcja Φ nie jest zdefiniowana dopóki BAL nie wykona pierwszego przesunięcia serwera (załóżmy, że zachodzi to w kroku t_0). Dodatkowo w przeciwieństwie do poprzednio rozważanych funkcji potencjału Φ

[5] $D(v_0)$ było niezdefiniowane, na końcu kroku $D'(v_\ell)$ jest niezdefiniowane.

niekoniecznie jest nieujemna. Jednak na mocy lematu G.2, Φ jest ograniczona od dołu przez wielkości niezależne od sekwencji wejściowej. Zatem istnieje takie F_1 (będące funkcją odległości punktów w metryce), że $\Phi \geq F_1$.

Koszt algorytmu BAL do momentu pierwszego przesunięcia serwera jest ograniczony; oznaczmy to ograniczenie przez F_2. Jeśli pokażemy, że w każdym kroku $t \geq t_0$ zachodzi

$$\textsc{Bal}(t) + \Delta\Phi(t) \leq k \cdot \textsc{Opt}(t) \ , \tag{16}$$

to otrzymamy, że dla dowolnej sekwencji wejściowej σ zachodzi

$$\textsc{Bal}(\sigma) \leq k \cdot \textsc{Opt}(\sigma) + F_2 - F_1 \ ,$$

czyli że algorytm BAL jest k-konkurencyjny (choć niekoniecznie ściśle k-konkurencyjny).

Zauważmy, że jeśli odwołanie jest w wierzchołku różnym od dziury CURR, to BAL nie przesuwa żadnego z serwerów, jego koszt jest równy zero, wierzchołki CURR, PREV i wartości $D(\cdot)$ pozostają takie same, a więc $\Delta\Phi = 0$. Koszt algorytmu optymalnego jest oczywiście nieujemny, a zatem nierówność 16 jest trywialnie spełniona.

Poniżej będziemy zatem zakładać, że żądanie występuje w wierzchołku CURR. Rozbijemy (tak, znowu) operacje w kroku $t \geq 2$ na dwie akcje.

Akcja 1, ruch OPT. Jeśli na początku ruchu OPT nie ma dziury w wierzchołku w wierzchołku CURR, to ponieważ jest algorytmem leniwym, to nic nie robi, potencjał pozostaje niezmieniony i nierówność 16 jest trywialnie spełniona. Załóżmy więc, że OPT przesuwa jeden ze swoich serwerów z wierzchołka v_j do wierzchołka CURR, płacąc $d(v_j, \textsc{curr})$. Pokażemy, że potencjał wzrasta co najwyżej o $k \cdot d(v_j, \textsc{curr})$.

Przed ruchem potencjał jest równy $k \cdot (D(\textsc{prev}) - d(\textsc{curr}, \textsc{prev})) - S$, natomiast po ruchu jest równy $k \cdot D(v_j) - S$. Zatem

$$\Delta\Phi = k \cdot \left(D(v_j) - D(\textsc{prev}) + d(\textsc{curr}, \textsc{prev}) \right) \leq k \cdot d(v_j, \textsc{curr}) \ ,$$

gdzie ostatnia nierówność wynika z obserwacji C.2.

Akcja 2, ruch BAL. Na początku tej akcji OPT ma już serwer w punkcie CURR (a jego dziura — używając oznaczenia z poprzedniego przypadku — jest w punkcie v_j) Stąd potencjał na początku tej akcji wynosi $\Phi_{\mathrm{B}} = k \cdot D(v_j) - S$. Niech S' oznacza wartość S po ruchu BAL; łatwo zauważyć, że $S' = S + d(v_\ell, \textsc{curr})$, gdzie v_ℓ jest pozycją serwera, który jest przesuwany do wierzchołka CURR. Z tego przesunięcia wynika, że

$$D'(\textsc{curr}) = D(v_\ell) + d(v_\ell, \textsc{curr}) \ .$$

Dodatkowo po przesunięciu wierzchołki PREV i CURR zmieniają się:

$$\textsc{curr}' = v_\ell \qquad\qquad \textsc{prev}' = \textsc{curr}$$

Jeśli $v_\ell = v_j$, nowa dziura algorymu BAL tworzy się w tym samym miejscu co dziura OPT. Wtedy potencjał na końcu ruchu wynosi

$$\begin{aligned} \Phi_{\mathrm{F}} &= k \cdot (D'(\textsc{prev}') - d(\textsc{curr}', \textsc{prev}')) - S' \\ &= k \cdot (D'(\textsc{curr}) - d(v_\ell, \textsc{curr})) - S' \\ &= k \cdot D(v_\ell) - (S + d(v_\ell, A)) \\ &= \Phi_{\mathrm{B}} - d(v_\ell, A) \ . \end{aligned}$$

W drugim przypadku $v_\ell \neq v_j$. Wtedy potencjał na końcu wynosi $\Phi_{\mathrm{F}} = k \cdot D(v_j) - S'$, czyli również jest mniejszy od Φ_{B} o $d(v_\ell, A)$.

Pokazaliśmy więc, że zmniejszenie się potencjału pokrywa koszt algorytmu.

Zatem w każdym przypadku nierówność 16 zachodzi. ■

D. Routing: Optymalizacja obciążenia

Dla dowolnej krawędzi e definiujemy *obciążenie krawędzi* $B(e) = |\{j : e \in P_j\}|$ jako liczbę przechodzących przez nią scieżek.

> Optymalizacja obciążenia. Dany jest graf $G = (V, E)$. Wejście składa się z par (s_j, t_j), dla których algorytm wybiera ścieżkę P_j. Celem jest minimalizacja maksymalnego obciążenia krawędzi, tj. $\max_e B(e)$. Tę ostatnią wielkość nazywamy po prostu *obciążeniem*.

Jeśli popatrzymy na ten problem w sposób przyrostowy, to chcielibyśmy śledzić wartość naszego rozwiązania (czyli obecnie maksymalnego obciążenia) i patrzeć w jaki sposób zmieni się ono jeśli wybierzemy kolejna ścieżkę. Chcielibyśmy, żeby wartość tego rozwiązania nie wzrastała wtedy albo żeby wzrastała o mały składnik, który jesteśmy w stanie porównać do wzrostu kosztu rozwiązania optymalnego.

Zauważmy jednak, że jeśli chcemy optymalizować tylko jedną konkretną wartość (maksymalne obciążenie), to gubimy informację na temat struktury obecnego rozwiązania. Przede wszystkim tę samą wartość maksymalnego obciążenia L osiągamy jeśli obciążenie jedej krawędzi wynosi L jak i kiedy obciążenie wszystkich krawędzi wynosi L. Dodatkowo bardzo trudno jest dobrze uzależnić wzrost maksymalnego obciążenia od jego aktualnej wielkości i wyboru ścieżki w danym kroku.

Zauważmy teraz, że jeśli dla dowolnego $a > 1$ weźmiemy wielkość $\log_a \sum_e a^{B(e)}$ to zachodzi $\max_e\{B(e)\} \leq \log_a \sum_e a^{B(e)} \leq \log_a m + \max_e\{B(e)\}$, gdzie m jest liczbą krawędzi w grafie. Zarazem wielkość $\sum_e a^{B(e)}$ (będziemy ją nazywać *wagą rozwiązania*) niesie pewną informację o strukturze obecnego rozwiązania, którą będziemy w stanie wykorzystać.

Jako pierwsze przybliżenie rozwiązania, ilustrujące naszą metodę pokażemy algorytm, który minimalizuje obciążenie pod warunkiem, że optymalne rozwiązanie generuje obciążenie 1.

Twierdzenie D.1. *Istnieje algorytm* Exp$_1$*, który dla dowolnej sekwencji* σ*, takiej że* Opt$(\sigma) = 1$*, generuje ociążenie* $\mathcal{O}(\log m)$.

Dowód. Parametrem algorytmu jest pewne ustalone $a \in (1, 2)$. Załóżmy, że aktualne obciążenie na początku kroku j opisywane jest przez funkcję B_{j-1}. Algorytm jest zachłanny: ścieżka P_j jest wybierana tak, żeby zminimalizować wartość

$$X_j := \sum_{e \in P_j} \left(a^{B_{j-1}(e)+1} - a^{B_{j-1}(e)} \right) \ .$$

Jak taką ścieżkę wyliczyć w czasie wielomianowym od m pozostawiamy jako proste ćwiczenie. Zauważmy, że X_j jest różnicą pomiędzy wagą rozwiązania w kolejnych dwóch krokach: $X_j = \sum_e (a^{B_j(e)} - a^{B_{j-1}(e)})$.

Dla dowodu lematu weźmy dowolną sekwencję wejściową σ długości N, taką że Opt$(\sigma) = 1$. Kolejne X_j sumują się teleskopowo i dostajemy

$$\sum_{j=1}^{N} X_j = \sum_e a^{B_N(e)} - \sum_e a^{B_0(e)} = \sum_e \left(a^{B_N(e)} \right) - m \ . \tag{17}$$

Jak ograniczyć X_j? Niech P_j^* będzie ścieżką wybraną w j-tym kroku przez algorytm optymalny.

$$\begin{aligned} X_j &= \sum_{e \in P_j} \left(a^{B_{j-1}(e)+1} - a^{B_{j-1}(e)} \right) \\ &\leq \sum_{e \in P_j^*} \left(a^{B_{j-1}(e)+1} - a^{B_{j-1}(e)} \right) && \text{(z zachłanności algorytmu)} \\ &= \sum_{e \in P_j^*} a^{B_{j-1}(e)} \cdot (a-1) \\ &\leq \sum_{e \in P_j^*} a^{B_N(e)} \cdot (a-1) \ . \end{aligned}$$

Sumując po wszystkich krokach otrzymujemy:

$$\begin{aligned} \sum_{j=1}^{N} X_j &\leq (a-1) \cdot \sum_{j=1}^{N} \sum_{e \in P_j^*} a^{B_N(e)} \\ &= (a-1) \cdot \sum_e \sum_{j \,:\, e \in P_j^*} a^{B_N(e)} \end{aligned}$$

Ponieważ założyliśmy, że rozwiązanie Opt generuje obciążenie co najwyżej 1, zatem wewnętrzna suma wynosi co najwyżej $a^{B_N(e)}$:

$$\sum_{j=1}^{N} X_j \leq (a-1) \cdot \sum_e a^{B_N(e)} \ . \tag{18}$$

Porównując nierówności (17) i (18) otrzymujemy $(2-a) \cdot \sum_e a^{B_N(e)} \leq m$. Stąd dla każdej krawędzi e zachodzi $a^{B_N(e)} \leq \frac{m}{2-a}$, a zatem $B_N(e) = \mathcal{O}(\log m)$. ■

Teraz rozszerzymy powyższy dowód na bardziej realistyczne przypadki.

Twierdzenie D.2. *Dla dowolnej sekwencji wejściowej, którą algorytm optymalny jest w stanie obsłużyć z obciążeniem nie większym niż λ, istnieje algorytm online Exp$_\lambda$, który generuje obciążenie $\mathcal{O}(\log m) \cdot \lambda$.*

Dowód. Zmodyfikujmy lekko algorytm. Tym razem w kroku j algorytm chce tak wybrać ścieżkę P_j, żeby minimalizować wartość $X_j := \sum_{e \in P_j} a^{(B_{j-1}(e)+1)/\lambda} - \sum_e a^{B_{j-1}(e)/\lambda}$. Prześledźmy, w których miejscach trzeba zmodyfikować dowód poprzedniego twierdzenia. Równanie (17) przybiera postać

$$\sum_{j=1}^{N} X_j = \sum_e \left(a^{B_N(e)/\lambda} \right) - m \ , \tag{19}$$

natomiast ograniczenie na X_j jest obecnie równe

$$X_j \leq \sum_{e \in P_j^*} a^{B_N(e)/\lambda} \cdot (a^{1/\lambda} - 1) \ .$$

Jak poprzednio sumując po wszystkich krokach otrzymujemy

$$\sum_{j=1}^{N} X_j \leq (a^{1/\lambda} - 1) \cdot \sum_e \sum_{j \,:\, e \in P_j^*} a^{B_N(e)/\lambda} \ .$$

TODO tutaj skorzystac z nierownosci Poniżej wykorzystamy nierówność $a^x - 1 \leq (a-1) \cdot x$, która jest prawdziwa dla dowolnego $x \in [0,1]$.

Podobnie jak poprzednio, ponieważ Opt $\leq \lambda$, to $|\{j \,:\, e \in P_j^*\}| \leq \lambda$ dla dowolnej krawędzi e. Zatem nowa postać nierówności (18) jest następująca:

$$\begin{aligned} \sum_{j=1}^{N} X_j &\leq \lambda \cdot (a^{1/\lambda} - 1) \cdot \sum_e a^{B_N(e)/\lambda} \\ &\leq (a-1) \cdot \sum_e a^{B_N(e)/\lambda} \qquad \text{(bo } a^x - 1 \leq (a-1) \cdot x \text{ dla } x \in [0,1]) \ . \end{aligned} \tag{20}$$

Porównując ze sobą nierówności (19) i (20) otrzymujemy $B_N(e) = \mathcal{O}(\log m) \cdot \lambda$. ■

E. Rozłączne ścieżki na prostej: dolne ograniczenie

Lemat E.1 (wariant min-max Yao dla ścisłej konkurencyjności)**.** *Rozważmy dowolny problem maksymalizacyjny. Załóżmy, że istnieje rozkład prawdopodobieństwa π nad zbiorem wszystkich możliwych sekwencji wejściowych $\mathcal{I}$, taki że dla dowolnego algorytmu deterministycznego* Det*, jeśli wybierzemy wejście $\sigma \in \mathcal{I}$ zgodnie z tym rozkładem, zachodzi $\mathbf{E}_\pi[Det(\sigma)] \leq \frac{1}{\mathcal{R}} \cdot \mathbf{E}_\pi[Opt(\sigma)]$. Wtedy współczynnik ścisłej konkurencyjności dowolnego zrandomizowanego algorytmu dla tego problemu wynosi co najmniej $\mathcal{R}$.*

Dowód. Weźmy dowolny algorytm zrandomizowany Alg i załóżmy nie wprost, że osiąga on współczynnik konkurencyjności $\mathcal{R}' < R$. Algorytm ten jest równoważny pewnemu rozkładowi prawdopodobieństwa $\mathcal{A}$ nad wszystkimi algorytmami deterministycznymi. Dla każdego σ mamy zatem $\mathbf{E}_\mathcal{A}[\textsc{Det}(\sigma)] \geq \frac{1}{\mathcal{R}'} \cdot \textsc{Opt}(\sigma) > \frac{1}{\mathcal{R}} \cdot \textsc{Opt}(\sigma)$. Dlatego uśredniając po losowych wejściach σ

$$\mathbf{E}_\pi \mathbf{E}_\mathcal{A}[\textsc{Det}(\sigma)] > \frac{1}{\mathcal{R}} \cdot \mathbf{E}_\pi[\textsc{Opt}(\sigma)] \ .$$

Jednocześnie z założenia lematu otrzymujemy

$$\mathbf{E}_\mathcal{A} \mathbf{E}_\pi[\textsc{Det}(\sigma)] \leq \frac{1}{\mathcal{R}} \cdot \mathbf{E}_\mathcal{A} \mathbf{E}_\pi[\textsc{Opt}(\sigma)] = \frac{1}{\mathcal{R}} \cdot \mathbf{E}_\pi[\textsc{Opt}(\sigma)] \ .$$

Ponieważ w powyższych dwóch wzorach możemy zamienić miejscami $\mathbf{E}_\mathcal{A}$ z $\mathbf{E}_\pi$, lewe strony nierówności są sobie równe i otrzymujemy sprzeczność. ■

Twierdzenie E.2. *Żaden algorytm zrandomizowany Alg dla problemu rozłącznych ścieżek na prostej nie może osiągać współczynnika konkurencyjności mniejszego niż $\lfloor \frac{\log N}{2} \rfloor$.*

Dowód. Dla uproszczenia załóżmy znowu, że $N = 2^k$. Skonstruujemy rozkład prawdopodobieństwa π nad możliwymi sekwencjami wejściowymi σ, taki że dla dowolnego algorytmu deterministycznego Det zachodzi

$$\mathbf{E}[\textsc{Opt}(\sigma)] = \frac{\log N}{2} \qquad \text{oraz} \qquad \mathbf{E}[\textsc{Det}(\sigma)] \leq 1 \ ,$$

gdzie wartość oczekiwana brana jest po rozkładzie π. Z zasady min-max Yao dostaniemy wtedy tezę twierdzenia.

Rozważmy następujące zbiory połączeń, gdzie $[a, b]$ oznacza połączenie od a do b.

$$\begin{aligned} C_1 &= \{[1, N]\} \\ C_2 &= \{[1, N/2], [N/2 + 1, N]\} \\ &\vdots \quad \vdots \\ C_{\log N} &= \{[1, 2], [3, 4], [5, 6], \ldots, [N - 1, N]\} \end{aligned}$$

Oczywiście, wszystkie połączenia ze zbioru C_i są połączeniami poziomu i. Wejście σ jest generowane w następujący sposób. Na początku ℓ jest wybierane ze zbioru $\{1, 2, \ldots, \log N\}$ z prawdopodobieństwem $\frac{2^{-\ell}}{1-1/N}$. Następnie sekwencja składa się ze wszystkich połączeń ze zbiorów $C_1, C_2, \ldots, C_\ell$.

Najpierw obliczymy oczekiwany zysk Opt na takiej losowej sekwencji. Jeśli wejście kończy się połączeniami poziomu i, to optymalne rozwiązanie akceptuje te właśnie połączenia odrzucając wszystkie poprzednie. Dlatego też

$$\mathbf{E}[\textsc{Opt}(\sigma)] \ = \ \sum_{i=1}^{\log N} 2^{i-1} \cdot \frac{2^{-i}}{1 - 1/N} \ > \ \frac{\log N}{2} \ .$$

Pokażemy teraz, że dla dowolnej deterministycznej strategii Det, $\mathbf{E}[\textsc{Det}(\sigma)] \leq 1$. W naturalny sposób połączenia ze zbiorów C_i tworzą pełne drzewo binarne, w którego korzeniu jest połączenie $[1, N]$ a w $N/2$ liściach elementy ze zbioru $C_{\log N}$. Dla węzła r odpowiadające mu połączenie oznaczamy przez c_r. Mówimy, że węzeł drzewa jest aktywny, jeśli odpowiadające mu połączenie występuje w sekwencji wejściowej; Dla węzła r definiujemy zmienną losową X_r określającą zysk algorytmu wynikający z zaakceptowanych połączeń występujących w poddrzewie o korzeniu w r, pod warunkiem że r jest wierzchołkiem aktywnym i pod warunkiem, że nie zostały zaakceptowane jeszcze połączenia kolidujące z c_r. Pokażemy, że dla dowolnego r, $\mathbf{E}[X_r] \leq 1$. Ponieważ korzeń całego drzewa jest wierzchołkiem aktywnym będzie z tego wynikać, że $\mathbf{E}[\textsc{Det}(\sigma)] \leq 1$.

Dowód będzie przez indukcję względem poziomu drzewa. Jeśli aktywny jest liść r, to algorytm może albo odrzucić c_r (co jest śmiałym i dość nierozsądnym posunięciem), wtedy $X_r = 0$ lub przyjąć to połączenie, wtedy $X_r = 1$.

Załóżmy teraz, że aktywny jest węzeł r naszego drzewa i niech r_1 i r_2 oznaczają jego synów. Jeśli algorytm przyjmie połączenie c_r to $X_r = 1$. Jeśli algorytm odrzuci to połączenie (w nadziei, że dostanie więcej na niższych poziomach) to z definicji rozkładu π wynika, że wierzchołki r_1 i r_2 są aktywne z prawdopodobieństwem mniejszym $1/2$. Formalnie mamy

$$\mathbf{Pr}[\ell \geq i+1 | \ell \geq i] \;=\; \frac{\mathbf{Pr}[\ell \geq i+1]}{\mathbf{Pr}[\ell \geq i]} \;=\; \frac{\frac{1}{2^{i+1}} + \frac{1}{2^{i+2}} + \ldots + \frac{1}{2^{\log N}}}{\frac{1}{2^{i}} + \frac{1}{2^{i+1}} + \ldots + \frac{1}{2^{\log N}}} \;<\; \frac{1}{2}\ .$$

Zauważmy, że przy odrzuceniu c_r nie ma połączeń kolidujących z c_{r_1} ani z c_{r_2}. Zatem wtedy $\mathbf{E}[X_r] < \frac{1}{2} \cdot \mathbf{E}[X_{r_1}] + \frac{1}{2} \cdot \mathbf{E}[X_{r_2}]$. Z założenia indukcyjnego dostajemy, że w tym wypadku $\mathbf{E}[X_r] \leq 1$, co kończy dowód indukcyjny. ■

F. Lepsza analiza algorytmu dla pakowania pojemników

Jednym z najbardziej naturalnych algorytmów dla tego problemu jest algorytm First-Fit (FF). Algorytm ten zakłada, że pojemniki są ponumerowane liczbami naturalnymi i wkłada dany przedmiot do pierwszego pojemnika, do którego włożenie jest możliwe.

Twierdzenie F.1. *Algorytm First-Fit jest* $7/4$*-konkurencyjny.*

Dowód. Ponownie weźmy dowolny ciąg wejściowy σ zawierający m przedmiotów. Niech B oznacza zbiór wykorzystanych przez First-Fit pojemników, $|B| = k$. Podzielmy go na trzy pozbiory B_1, B_2 i B_3, i niech $k_i = |B_i|$.

B_1**:** pojemniki zawierające jeden przedmiot o wadze większej niż $1/2$;

B_2**:** pojemniki zawierające dwa lub więcej przedmiotów o sumarycznej wadze większej niż $2/3$;

B_3**:** pozostałe pojemniki.

Podobnie jak powyżej można pokazać, że $k_3 \leq 2$. W tym celu zauważmy, że w trzecim zbiorze może być co najwyżej jeden pojemnik zawierający pojedynczy przedmiot o wadze nie większej niż $1/2$.[6] Z definicji B_3 poza rozpatrzonymi powyżej pojemnikami mogą się tam jeszcze znajdować pojemniki, które mają dwa lub więcej przedmiotów o sumarycznej wadze nie większej niż $2/3$; wystarczy pokazać, że jest co najwyżej jeden taki pojemnik. Załóżmy nie wprost, że są dwa takie pojemniki b_i i b_j (gdzie $i \leq j$). W pojemniku b_j są co najmniej dwa przedmioty, więc jeden z nich ma wagę nie większą niż $1/3$. Zatem mógł on zostać dołożony do pojemnika b_i.

Stąd dostajemy, że $\mathrm{FF}(\sigma) \leq k_1 + k_2 + 2$. Z drugiej strony $\textsc{Opt}(\sigma) \geq k_1$ (bo każdy z przedmiotów o wadze większej niż $1/2$ musi być w osobnym pojemniku) oraz $\textsc{Opt}(\sigma) \geq w(B_1) + w(B_2) \geq \frac{1}{2} \cdot k_1 + \frac{2}{3} \cdot k_2$. Można pokazać, że dla dowolnych $k_1, k_2 \geq 0$, $k_1 + k_2 > 0$ zachodzi

$$\frac{k_1 + k_2}{\max\{k_1, \frac{1}{2} \cdot k_1 + \frac{2}{3} \cdot k_2\}} \leq \frac{7}{4} \ ,$$

a zatem

$$\mathrm{FF}(\sigma) \leq \frac{7}{4} \cdot \textsc{Opt}(\sigma) + 2 \ . \qquad \blacksquare$$

[6] Jeśli są dwa takie pojemniki, to dojdziemy do sprzeczności jak w dowodzie poprzedniego twierdzenia.

G. Szeregowanie zadań

Choć szeregowanie zadań można rozpatrywać jako inny problem pakowania pojemników, przyjrzymy się definicji tego problemu w oryginalnym kontekście.

> SZEREGOWANIE ZADAŃ. Danych jest m identycznych procesorów. Sekwencja wejściowa składa się z zadań o określonym czasie wykonania. Należy przyporządkować dane zadanie do jednego z procesorów. Niech ℓ_i będzie sumą zadań przypisanych do procesora i. Chcemy, żeby $\max_i\{\ell_i\}$ było jak najmniejsze.

Twierdzenie G.1. *Algorytm* GREEDY *dla problemu szeregowania zadań na m procesorach jest $(2-1/m)$-konkurencyjny*

Dowód. Weźmy dowolny ciąg wejściowy σ. Bez straty ogólności możemy założyć, że najbardziej obciążona będzie pierwszy procesor. Niech w oznacza wagę ostatniego zadania na pierwszym procesorze, a s sumaryczną wagę pozostałych zadań przypisanych do tego procesora. Obciążenie każdego innego procesora musi również wynosić co najmniej s, bo inaczej w zostałoby przypisane do innego procesora. Zatem sumaryczna waga wszystkich zadań to $w + s \cdot m$ i stąd

$$\textsc{Opt}(\sigma) \geq \frac{w + s \cdot m}{m} = \frac{w}{m} + s \ .$$

Jednocześnie $\textsc{Opt}(\sigma) \geq w$. Zatem:

$$\begin{aligned} \textsc{Greedy}(\sigma) &= w + s \\ &\leq w + \textsc{Opt}(\sigma) - \frac{w}{m} \\ &\leq \textsc{Opt}(\sigma) + w \cdot \left(1 - \frac{1}{m}\right) \\ &\leq \textsc{Opt}(\sigma) \cdot \left(2 - \frac{1}{m}\right) \ . \end{aligned}$$

■

Rozważmy teraz zmodyfikowany problem szeregowania zadań. Zadanie i w sekwencji wejściowej ma przypisaną wagę w_i oraz zbiór procesorów $\mathcal{P}_i$. To zadanie można przyporządkowywać tylko do procesora ze zbioru $\mathcal{P}_i$.

Twierdzenie G.2. *Algorytm* GREEDY *dla problemu szeregowania ograniczonych zadań na m procesorach jest $(\lceil \log m \rceil + 1)$-konkurencyjny*

Dowód. Mamy m procesorów i wejście σ składające się z n zadań, gdzie i-te zadanie ma wagę w_i i zestaw dopuszczalnych procesorów $\mathcal{P}_i$. Algorytm zachłanny szereguje je w sposób przedstawiony na rysunku 4. Zapis „$w = x$ $(\to j)$" oznacza, że jest to zadanie o wadze x i zostało przyporządkowane w optymalnym rozwiązaniu do procesora j.

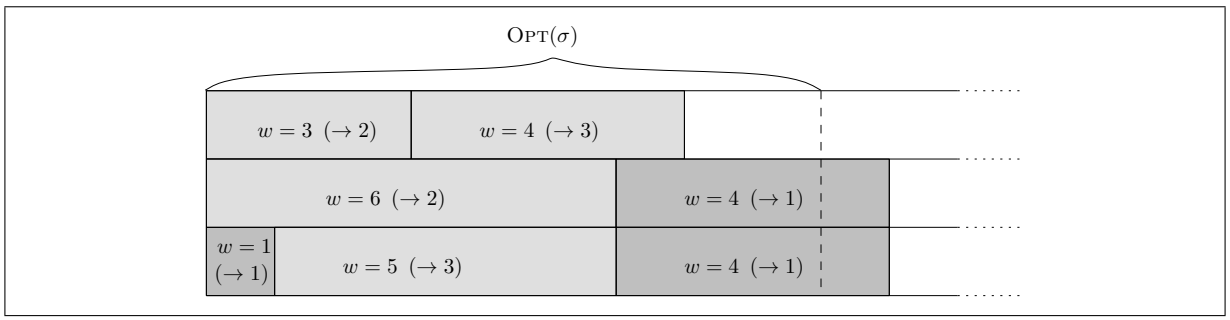
Dla potrzeb analizy dzielimy obciążenie na warstwy, każda z warstw ma grubość $\textsc{Opt}(\sigma)$. Wystarczy, że pokażemy, że wielkość obciążenia każdego z procesorów mieści się w $\lceil \log n \rceil + 1$ warstwach. W tym celu skoncentrujmy się na jednej warstwie (na rysunku jest to pierwsza warstwa); będziemy ją nazywać *bieżącą warstwą*, zbiór warstw które po niej następują będziemy nazywać *następnymi warstwami*, a zbiór warstw je poprzedzających nazywamy *poprzednimi warstwami*. Niech W_j będzie obciążeniem procesora j w bieżącej warstwie a R_j sumarycznym obciążeniem tego procesora w kolejnych warstwach. Oczywiście jeśli $W_j < \textsc{Opt}(\sigma)$ to $R_j = 0$. Poniżej pokażemy, że

$$\sum_{j=1}^{m} W_j \ \geq \ \sum_{j=1}^{m} R_j \ . \tag{21}$$

Aby udowodnić powyższą nierówność weźmy dowolny procesor k (na rysunku $k = 1$). Pokażemy, że część $\sum_{j=1}^{m} R_j$, która jest przeznaczona w rozwiązaniu optymalnym na procesor k (jej wielkość oznaczamy przez A_k) jest mniejsza bądź równa W_k. Ponieważ z definicji te części te są rozłączne i pokrywają zawartość następnych warstw dostaniemy, że $\sum_{k=1}^{m} W_k \geq \sum_{k=1}^{m} A_k = \sum_{j=1}^{m} R_j$.

Jak duże jest A_k? Niech O_k oznacza zbiór wszystkich zadań przeznaczonych w optymalnym rozwiązaniu na procesor k (ciemniejsze zadania na rysunku). Oczywiście $A_k \leq \sum_{j \in O_k} w_j \leq \textsc{Opt}(\sigma)$. Jeśli $W_k = \textsc{Opt}(\sigma)$, to nierówność $A_k \leq W_k$ jest trywialnie spełniona. Załóżmy więc, że $W_k < \textsc{Opt}(\sigma)$.

Aby obliczyć A_k, należy wziąć wartość $\sum_{j \in O_k} w_j$ i pomniejszyć ją o te zadania (lub ich części) z O_k, które leżą w bieżącej warstwie lub warstwach poprzednich. Zauważmy, że każde zadanie ze zbioru O_k musi w rozwiązaniu produkowanym przez GREEDY zaczynać się wcześniej niż w chwili W_k. W przeciwnym przypadku procesor k



Rysunek 4: Przykładowe szeregowanie zadań przez algorytm Greedy

zostałby wykorzystany dla obsługi tego zadania. Musimy od każdego zadania odjąć co najmniej $\textsc{Opt}(\sigma) - W_k$, a zatem otrzymujemy

$$A_k \leq \sum_{j \in O_k} w_j - (\textsc{Opt}(\sigma) - W_k) \leq W_k \ .$$

Teraz pokażemy jak z nierówności 21 wynika teza twierdzenia. Niech L_i oznacza sumę wag zadań w warstwie i i późniejszych. Wtedy $L_1 = \sum_{i=1}^{n} w_i$. Z (21) mamy, że

$$L_{i+1} \leq L_i/2 \ ,$$

a stąd

$$L_{\lceil \log m \rceil + 1} \leq \frac{L_1}{m} = \frac{\sum_{i=1}^{n} w_i}{m} \leq \textsc{Opt}(\sigma) \ .$$

Oznacza to, że w warstwie $\lceil \log m \rceil + 1$ mamy tak mało wagi do rozdysponowania między procesory, że dowolne przypisanie zadań (niekoniecznie zachłanne) gwarantuje zakończenie wykonywania zadań przed końcem tej warstwy. ■

H. Maksymalne skojarzenie online

MAKSYMALNE SKOJARZENIE. Mamy dane dwa równoliczne zbiory dziewczyn D i chłopców C, oba o mocy n. Wiemy, że graf połączeń między C i D jest dwudzielny, lecz jest nieznany na początku; połączenie oznacza, że dane dwie osoby o różnej płci się znają.

Wejście składa się z n kroków; w jednym kroku $1 \leq i \leq n$ poznajemy wszystkie krawędzie prowadzące od chłopca c_i do znanych mu dziewczyn. Należy wybrać jedną niewybraną jeszcze dziewczynę z którą c_i jest połączony; tę parę nazywamy skojarzoną. Należy zmaksymalizować moc zbioru skojarzeń.

Rozważmy algorytm zachłanny, tj. taki, który wybiera do skojarzenia dowolną krawędź prowadzącą od danego chłopca do jeszcze wolnej dziewczyny.

Twierdzenie H.1. *Przy założeniu, że w grafie istnieje doskonałe skojarzenie (o mocy n), powyższa zachłanna strategia jest 2-konkurencyjna.*

Dowód. Zauważmy, że algorytm zachłanny zwraca skojarzenie, które jest maksymalne, tj. nie daje się rozszerzyć. Dla instancji I dopuszczającej doskonałe skojarzenie $\textsc{Opt}(I) = n$. Zatem wystarczy pokazać, że $\textsc{Alg}(I) \geq n/2$.

Załóżmy, że tak nie jest. Oznacza to, że podczas działania algorytmu podzbiór chłopców C' został połączony ze zbiorem dziewczyn D', $|C'| = |D'| < n/2$ a pozostali chłopcy (ze zbioru $C \setminus C'$) nie zostali połączeni. Maksymalność skojarzenia implikuje, że w grafie mogą oni sąsiadować wyłącznie z już zajętymi dziewczynami (ze zbioru D').

Z drugiej strony $|C \setminus C'| > n/2$, a zatem z twierdzenia Halla[7] wynika, że zbiór sąsiadów tego zbioru musi mieć co najmniej tyle samo elementów. Otrzymana sprzeczność kończy dowód. ■

[7] Jak założyliśmy, w grafie istnieje skojarzenie doskonałe.

I. Porównanie adwersarzy

I.1. Silny adwersarz adaptujący się

Dotychczas rozważanego adwersarza, który nie widzi bitów losowych algorytmu nazywamy adwersarzem nieświadomym (ang. *oblivious*).

Adwersarz adaptujący się tworzy sekwencję wejściową na podstawie dotychczasowego zachowania algorytmu. Ponieważ algorytm może wykorzystywać bity losowe, a wejście jest (deterministyczną) funkcją zachowania algorytmu, wejście jest zmienną losową.

Rozróżniamy zatem dwa typy adwersarzy. *Silny adwersarz adaptujący się* oblicza rozwiązanie dla wygenerowanej sekwencji na końcu; możemy zatem założyć, że będzie to optymalne rozwiązanie dla wygenerowanej sekwencji. Mówimy zatem, że zrandomizowany algorytm ALG jest k-konkurencyjny przeciwko silnemu adwersarzowi adaptującemu się, jeśli istnieje taka stała α, że dla każdego w ten sposób generowanego wejścia I zachodzi

$$\mathbf{E}[\text{ALG}(I)] \leq k \cdot \mathbf{E}[\text{OPT}(I)] + \alpha \ ,$$

gdzie wartość oczekiwana jest liczona po wyborach losowych algorytmu.

Twierdzenie I.1. *Rozważmy dowolny algorytm ALG dla SRP. Jego współczynnik ścisłej konkurencyjności przeciwko silnemu adwersarzowi adaptującemu się wynosi co najmniej $\mathcal{R} = 2 - 1/B$.*

Dowód. Dowód jest identyczny jak dla przypadku dolnego ograniczenia dla deterministycznego algorytmu. Sekwencja σ tworzona przez adwersarza zawsze kończy się w dniu, w którym algorytm decyduje się na zakup nart. Załóżmy, że następuje to dniu $T \in \mathbb{N} \cup \{\infty\}$, gdzie T jest zmienną losową zależną od wyborów losowych algorytmu.

Wtedy $\text{ALG}(\sigma) = T - 1 + B$. Z drugiej strony $\text{OPT}(\sigma) = \min\{B, T\}$. Wystarczy udowodnić, że $\mathbf{E}[\text{ALG}(\sigma)] \geq \mathcal{R} \cdot \mathbf{E}[\text{OPT}(\sigma)]$. My potrafimy udowodnić silniejsze twierdzenie, a mianowicie

$$\text{ALG}(\sigma) \geq \mathcal{R} \cdot \text{OPT}(\sigma) \ .$$

Powyższa nierówność zmiennych losowych jest spełniona dla każdego wyboru zdarzenia elementarnego. Wynika ona z prostej analizy dwóch przypadków ($B \geq T$ lub $B < T$). ■

Twierdzenie I.2. *Rozważmy problem w którym ALG ma zawsze skończony zbiór odpowiedzi. Jeśli ALG jest R-konkurencyjny przeciwko silnemu adwersarzowi adaptującemu się, to istnieje R-konkurencyjny deterministyczny algorytm.*

Adwersarz adaptujący się musi generować rozwiązanie dla danej sekwencji w sposób *online*, wraz z algorytmem. Myślimy o tym, że adwersarz dostarcza swój własny deterministyczny algorytm ADV, który jest funkcją dotychczas widzianego kawałka wejścia i przeszłych akcji algorytmu ALG (tj. decyzję w kroku n może podjąć na podstawie odpowiedzi algorytmu do kroku $n-1$). Mówimy, że algorytm ALG jest k-konkurenyjny przeciwko adwersarzowi adaptującemu się, jeśli istnieje taka stała α, że dla każdego możliwego algorytmu adwersarza ADV i każdego generowanego w taki sposób wejścia I zachodzi

$$\mathbf{E}[\text{ALG}(I)] \leq k \cdot \mathbf{E}[\text{ADV}(I)] + \alpha \ ,$$

gdzie wartość oczekiwana jest liczona po wyborach losowych algorytmu.

Jeśli weźmiemy dowolny algorytm ALG i przez $\mathcal{R}_{\text{N}}(\text{ALG})$, $\mathcal{R}_{\text{A}}(\text{ALG})$ i $\mathcal{R}_{\text{SA}}(\text{ALG})$ oznaczymy osiągane przez niego współczynniki konkurencyjności przeciwko adwersarzowi nieświadomemu, adaptującemu się i silnemu adaptującemu się, to zachodzi:

$$\mathcal{R}_{\text{N}}(\text{ALG}) \leq \mathcal{R}_{\text{A}}(\text{ALG}) \leq \mathcal{R}_{\text{SA}}(\text{ALG}) \ .$$

Druga nierówność jest oczywista, zaś pierwsza wynika z tego, że adwersarz adaptujący się jest w stanie symulować dowolnego adwersarza nieświadomego generując sekwencję i rozwiązanie dla niej na samym początku a następnie ignorując faktyczne zachowanie algorytmu.

Powyższa nierówność zachodzi dla każdego algorytmu randomizowanego. Weźmy zatem dowolny problem online. Niech współczynniki konkurencyjności najlepszych algorytmów zrandomizowanych rozwiązujących ten problem to odpowiednio $\mathcal{R}_{\text{N}}$, $\mathcal{R}_{\text{A}}$ i $\mathcal{R}_{\text{SA}}$, a współczynnik konkurencyjności najlepszego algorytmu deterministycznego to $\mathcal{R}_{\text{DET}}$. Wtedy otrzymujemy

$$\mathcal{R}_{\text{N}} \leq \mathcal{R}_{\text{A}} \leq \mathcal{R}_{\text{SA}} \leq \mathcal{R}_{\text{DET}} \ . \tag{22}$$

Twierdzenie I.3. *Rozważmy dowolny algorytm* Alg *dla SRP. Jego współczynnik ścisłej konkurencyjności przeciwko adwersarzowi adaptującemu się wynosi co najmniej* $\mathcal{R} = 2 - 1/B$.

Dowód. Wprowadzamy takie same oznaczenia jak w poprzednim dowodzie. Jak poprzednio adwersarz będzie tworzyć wejście σ do dnia, w którym algorytm kupi narty.

Ten dowód jest trochę trudniejszy niż poprzedni, gdyż adwersarz musi zdecydować o swojej strategii w sposób *online*, nie znając wartości zmiennej losowej T. Jednak na podstawie kodu algorytmu adwersarz może wyliczyć wartość $\mathbf{E}[T]$. Jeśli $\mathbf{E}[T] \geq B$, to strategia adwersarza brzmi „kup narty pierwszego dnia", a w przeciwnym przypadku strategia to „zawsze pożyczaj narty".

W pierwszym z przypadków otrzymujemy

$$\mathbf{E}[\textsc{Alg}(\sigma)] \;=\; \mathbf{E}[T] - 1 + B \;\geq\; 2B - 1 \;\geq\; \mathcal{R} \cdot B \;=\; \mathcal{R} \cdot \textsc{Adv}(\sigma) \;.$$

Natomiast w drugim

$$\mathbf{E}[\textsc{Alg}(\sigma)] \;=\; \mathbf{E}[T] - 1 + B \;=\; \frac{\mathbf{E}[T] - 1 + B}{\mathbf{E}[T]} \cdot \mathbf{E}[T] \;\geq\; \frac{B - 1 + B}{B} \cdot \mathbf{E}[T] \;\geq\; \mathcal{R} \cdot \mathbf{E}[\textsc{Adv}(\sigma)] \;,$$

co kończy dowód. ∎

W tej części połączymy konkurencyjność algorytmów przeciwko różnym typom adwersarzy w sposób bardziej precyzyjny niż ten dany przez nierówności (22). W.13A

Dla skrócenia zapisu, w tym rozdziale rozważymy definicję konkurencyjności zadaną przez funkcje liniowe. Niech γ będzie funkcją liniową. Jeśli mówimy, że deterministyczny algorytm Alg jest γ-konkurencyjny, to tozumiemy przez to że dla dowolnego wejścia I, $\textsc{Alg}(I) \leq \gamma(\textsc{Opt}(I))$. W analogiczny sposób definiujemy konkurencyjność algorytmów zrandomizowanych; zauważmy, że funkcje liniowe są przemienne z wartością oczekiwaną.

I.2. Związek między adwersarzami przeciwko algorytmom zrandomizowanym

Na początku przyjrzymy się bliżej adwersarzom adaptującym się. Adwersarz taki składa się z dwóch części: deterministycznej części Q dającej zapytania i deterministycznej części odpowiadającej na te zapytania S, równolegle do algorytmu. Wtedy algorytm jest γ-konkurencyjny przeciwko adwersarzowi adaptującemu się, jeśli jest γ-konkurencyjny przeciwko dowolnemu adwersarzowi (Q, S) gdzie S jest algorytmem online, oraz jest γ-konkurencyjny przeciwko silnemu adwersarzowi adaptującemu się, jeśli jest γ-konkurencyjny przeciwko dowolnemu adwersarzowi $(Q, \textsc{Opt})$.

Twierdzenie I.4. *Niech* Alg *będzie zrandomizowanym algorytmem, który jest γ-konkurencyjny przeciwko adwersarzowi adaptującemu się, a* Alg′ *algorytmem, który jest δ-konkurencyjny przeciwko adwersarzowi nieświadomemu. Wtedy* Alg *jest $(\gamma \circ \delta)$-konkurencyjny przeciwko silnemu adwersarzowi adaptującemu się.*

Dowód. Niech $(Q, \textsc{Opt})$ będzie dowolnym adwersarzem adaptive-offline. Pokażemy, że Alg jest przeciwko niemu $(\gamma \circ \delta)$-konkurencyjny.

Najpierw pokażemy że Alg jest γ-„konkurencyjny" przeciwko adwersarzowi $(Q, \textsc{Alg}')$. Konkurencyjność jest tutaj w cudzysłowie, gdyż Alg′ jest algorytmem zrandomizowanym i nie zdefiniowaliśmy co to znaczy. Alg′ jest pewnym rozkładem π nad wszystkimi algorytmami deterministycznymi $\{S_j\}_j$, z kolei Alg jest pewnym innym rozkładem prawdopodobieństwa nad wszystkimi algorytmami deterministycznymi $\{\textsc{Det}_i\}_i$.

Ponieważ Alg jest γ-konkurencyjny przeciwko dowolnemu adwersarzowi adaptującemu się, w szczególności jest γ-konkurencyjny przeciwko (Q, S_j) dla dowolnego j, tj.:

$$\mathbf{E}_i[\textsc{Det}_i(\sigma(Q, \textsc{Det}_i)] \leq \gamma(\mathbf{E}_i[S_j(\sigma(Q, \textsc{Det}_i)]) \;.$$

Mnożąc obie strony przez $\pi(j)$ i sumując po wszystkich j dostajemy

$$\begin{aligned}
\mathbf{E}_i[\textsc{Det}_i(\sigma(Q, \textsc{Det}_i)] \;&=\; \sum\nolimits_j \pi(j) \cdot \mathbf{E}_i[\textsc{Det}_i(\sigma(\textsc{Det}_i, Q))] \\
&\leq \sum\nolimits_j \pi(j) \cdot \gamma(\mathbf{E}_i[S_j(\sigma(\textsc{Det}_i, Q))]) \\
&= \mathbf{E}_j[\gamma(\mathbf{E}_i[S_j(\sigma(\textsc{Det}_i, Q))])] \\
&= \gamma(\mathbf{E}_i \mathbf{E}_j[S_j(\sigma(\textsc{Det}_i, Q))]) \;,
\end{aligned}$$

Z δ-konkurencyjności algorytmu Alg′ wynika, że $\mathbf{E}_j[S_j(\sigma)] \leq \delta(\textsc{Opt}(\sigma))$ zachodzi dla dowolnego wejścia σ, które nie zależy od wyborów losowych algorytmu Alg′. Zatem

$$\begin{aligned}
\mathbf{E}_i[\textsc{Det}_i(\sigma(\textsc{Det}_i, Q))] \;&\leq \gamma(\mathbf{E}_i[\delta(\textsc{Opt}(\sigma(\textsc{Det}_i, Q)))]) \\
&= (\gamma \circ \delta)\; (\mathbf{E}_i[\textsc{Opt}(\sigma(\textsc{Det}_i, Q))]) \;.
\end{aligned}$$

∎

I.3. Silny adwersarz adaptujący się a algorytmy deterministyczne

W tej części pokażemy, że randomizacja przeciwko silnemu adwersarzowi adaptującemu się nie pomaga. Do poniższego dowodu potrzebne jest jednak jedno techniczne założenie, że dla każdego żądania adwersarza zbiór możliwych odpowiedzi jest skończony.

Twierdzenie I.5. *Niech* Alg *będzie zrandomizowanym algorytmem, który jest γ-konkurencyjny przeciwko adwersarzowi adaptive-offline. Wtedy istnieje deterministyczny γ-konkurencyjny algorytm.*

Najpierw przyjrzymy się podejściu które nie działa. Ponownie traktujemy Alg jako rozkład prawdopodobieństwa nad wszystkimi algorytmami deterministycznymi Det_i. Z konkurencyjności algorytmu Alg wynika, że dla każdego adwersarza adaptive-offline (Q, Opt) dla dowolnego wejścia zachodzi $\mathbf{E}_i[\text{Det}_i(\sigma(\text{Det}_i, Q))] \leq \gamma(\mathbf{E}_i[\text{Opt}(\sigma(\text{Det}_i, Q))])$. Zatem

$$\mathbf{E}_i[\,\text{Det}_i(\sigma(\text{Det}_i, Q)) - \gamma(\text{Opt}(\sigma(\text{Det}_i, Q)))] \leq 0 \ .$$

Stąd wynika, że istnieje algorytm Det_i taki, że

$$\text{Det}_i(\sigma(\text{Det}_i, Q)) \leq \gamma(\text{Opt}(\sigma(\text{Det}_i, Q))) \ .$$

Niestety algorytm taki istnieje dla ustalonego Q; my potrzebujemy, żeby istniał jeden Det_j dobry dla wszystkich Q. Poniżej prezentujemy inne podejście do twierdzenia, tym razem skuteczne.

W.13B

Dowód twierdzenia I.5. Rozważmy drzewo gry pomiędzy adwersarzem a algorytmem. Zakładamy, że korzeń jest na poziomie 1 i reprezentuje pustą sekwencję wejściową. Krawędzie wychodzące z poziomów nieparzystych reprezentują możliwe zapytania, a krawędzie wychodzące z poziomów parzystych możliwe odpowiedzi algorytmu.

W tej terminologii Q jest po prostu funkcją, która dla każdego wierzchołka na poziomie nieparzystym wybiera wychodzącą z niego krawędź, a Alg definiuje dla każdego wierzchołka na poziomie parzystym rozkład prawdopodobieństwa nad wszystkimi wychodzącymi z niego krawędziami. Każdy wierzchołek jest identyfikowany przez parę (σ, a), gdzie σ jest wygenerowanym początkiem sekwencji wejściowej a a jest odpowiedzią algorytmu. Z wierzchołkiem takim wiążemy wartość $cost(\sigma, a)$, która jest równa kosztowi rozwiązania a na sekwencji σ.

Należący do nieparzystego poziomu wierzchołek (σ, a) nazywamy *natychmiast wygrywającym* (dla adwersarza) jeśli zachodzi dla niego $cost(\sigma, a) > \gamma(\text{Opt}(\sigma))$. Jeśli algorytm dojdzie do takiego wierzchołka, adwersarz natychmiast przerywa grę. Należący do nieparzystego poziomu wierzchołek (σ, a) nazywamy *wygrywającym* jeśli istnieje taka stała B, że przy optymalnej grze adwersarza każda ścieżka, którą może wybrać algorytm zawiera natychmiast wygrywający wierzchołek w ciągu co najwyżej B kroków. Powyższe definicje wierzchołków wygrywających i natychmiast wygrywających zależą tylko od struktury drzewa (i funkcji kosztu) a nie od konkretnych wartości Q i Alg.

Zauważmy, że korzeń drzewa nie może być wierzchołkiem wygrywającym, bo oznaczałoby to, że niezależnie od działań algorytmu adwersarz może wygenerować sekwencję, nie dłuższą niż B, która spowoduje, że algorytm skończy w wierzchołku natychmiast wygrywającym. Wtedy skonstruowanie γ-konkurencyjnego algorytmu byłoby niemożliwe, co przeczy założeniu.

Pokażemy teraz, że dla dowolnego niewygrywającego wierzchołka x dowolne żądanie adwersarza może być obsłużone (deterministycznie) tak, że algorytm skończy nadal w wierzchołku niewygrywającym. Indukcyjnie będzie to prowadzić do deterministycznego algorytmu, który będzie się poruszać zawsze po wierzchołkach niewygrywających, a zatem będzie γ-konkurencyjny. Załóżmy nie wprost, że istnieje syn wierzchołka x, y, taki że każdy syn wierzchołka y jest wygrywający. Ponieważ dla każdego syna y istnieje zatem ograniczenie na liczbę kroków która doprowadza do wierzchołka natychmiast wygrywającego i synów jest skończona ilość, istnieje też ograniczenie na liczbę kroków która prowadzi z wierzchołka x do wierzchołka natychmiast wygrywającego. Zatem x jest wygrywający, co przeczy założeniu. ■

J. Routing bez primal-dual

Niech $m = |E|$ Skonstruujemy algorytm $O(U)$-konkurencyjny, jeśli $U \geq \log 2m$. Da się go poprawić, żeby był $O(\log m)$ konkurencyjny.

Intuicyjnie chcemy wybierać takie ścieżki, które omijają przeciążone krawędzie. Z drugiej strony może to prowadzić do wyboru długich ścieżek. Nieopłacalne ścieżki będziemy odrzucać. Chcemy zatem optymalizować miarę $F_j = \sum_{e \in E} 2^{\ell_j(e)} - 1$.

Nasz algorytm będzie sprawdzać w kroku j czy istnieje ścieżka P od s_j do t_j, dla której zachodzi

$$\sum_{e \in P} 2^{\ell_{j-1}(e)} \leq m \ ,$$

W takim przypadku akceptuje połączenie wybierając powyższą ścieżkę, inaczej połączenie jest odrzucane. Zauważmy, że algorytm zaakceptuje pierwszą podaną ścieżkę. Poniżej T oznacza długość wejścia.

Lemat J.1. *Rozwiązanie generowane przez algorytm jest dopuszczalne.*

Dowód. Ustalmy krawędź e. Niech P_k będzie ostatnią ścieżkę, która przechodzi przez e. Ponieważ ścieżka ta została zaakceptowana zachodzi

$$\sum_{e \in P_k} 2^{\ell_{k-1}(e)} \leq m \ ,$$

a zatem w szczególności $\ell_{k-1}(e) \leq \log m$ i stąd $\ell_k(e) = \ell_{k-1}(e) + 1 \leq \log 2m \leq U$. ■

Zauważmy, że na początku $F_0 = 0$. Porównamy zysk Alg i Opt do F_T.

Lemat J.2. *Niech A będzie zbiorem połączeń zaakceptowanych przez Alg. Wtedy $m \cdot |A| \geq F_T$.*

Dowód. Zauważmy, że jeśli Alg odrzuca połączenie w kroku j to $F_j = F_{j-1}$. Natomiast jeśli Alg akceptuje połaczenie, to

$$F_j - F_{j-1} = \sum_{e \in P_j} \left(2^{\ell_j(e)} - 2^{\ell_{j-1}(e)}\right) = \sum_{e \in P_j} 2^{\ell_{j-1}(e)} \leq m \ .$$

W ostatniej nierówności skorzystaliśmy z definicji algorytmu. ■

Lemat J.3. *Niech B będzie zbiorem połączeń odrzuconych przez Alg ale zaakceptowanych przez Opt. Wtedy $m \cdot |B| \leq F_T \cdot U + U \cdot m$.*

Dowód. Weźmy dowolne połączenie $j \in B$, niech P_j^* będzie ścieżką użytą przez Opt. Ponieważ Alg odrzucił to żądanie, zachodzi

$$m < \sum_{e \in P_j^*} 2^{\ell_{j-1}(e)} \leq \sum_{e \in P_j^*} 2^{\ell_T(e)} \ .$$

Wysumujmy teraz powyższą nierówność po wszystkich $j \in B$. Ponieważ Opt jest rozwiązaniem dopuszczalnym, każda krawędź pojawi się w sumie co najwyżej U razy:

$$\begin{aligned} m \cdot |B| < \sum_{j \in B} \sum_{e \in P_j^*} 2^{\ell_T(e)} &= \sum_{e \in E} \sum_{j : e \in P_j^*} 2^{\ell_T(e)} \leq \sum_{e \in E} U \cdot 2^{\ell_T(e)} \\ &= \sum_{e \in E} U \cdot (2^{\ell_T(e)} - 1) + \sum_{e \in E} U \cdot 1 \\ &= U \cdot F_T + U \cdot m \ . \end{aligned}$$

■

Twierdzenie J.4. *Jeśli Algorytm jest ściśle $O(U)$-konkurencyjny.*

Dowód. Używamy $|A| \geq 1$. Mamy $\text{Opt} \leq |B| + |A| \leq U \cdot (F_t/m) + U + |A| \leq U \cdot |A| + U \cdot |A| + |A| = (2U + 1) \cdot |A| = (2U + 1) \cdot \text{Alg}$. ■

Dla $U \geq \log 2d$ można optymalizować $F_j = \sum_{e \in E} (4m)^{\ell_j(e)/U} - 1$ i cały dowód będzie prawie identyczny, ale na końcu dostaniemy $O(\log m)$-konkurejncyjność.